CSE 546 — Project 2 Report

Muskan Mehta (1225444701)

Sannidhya Pathania (1225329976)

Vaishnavi Amirapu (1225461211)

Table of contents

1. Problem Statement **1**
2. Design and implementation **1**
2.1 Architecture 1
Amazon Web services: 1
● AWS Lambda: 1
● Amazon DynamoDB: 1
● AWS S3: 1
● Amazon ECR: 2
Architecture Design: 2
2.2 Autoscaling 3
2.3 Member Tasks 3
3. Testing and evaluation **3**
4. Code **4**
4.1 Files 4
● Dockerfile 4
● encoding 4
● entry.sh 4
● handler.py 4
● mapping 4
● requirements.txt 4
4.2 Installation and Running 4
5. Individual contributions **6**
5.1 Muskan Mehta (1225444701) 6
5.2 Sannidhya Pathania (1225329976) 7
5.3 Vaishnavi Amirapu (1225461211) 8

1. Problem Statement

To build an elastic application using AWS PaaS cloud. The application should automatically scale in and scale out on demand and cost-effectively. AWS Lambda, which is a function-based serverless computing service should be used along with other supporting services from AWS. The application should implement a smart classroom assistant for educators. The application should take videos from the user's classroom and perform face recognition on the collected videos. Then it should look up the recognized students in the database. In the end, it should return the relevant academic information of each student back to the user.

2. Design and implementation

2.1 Architecture

Amazon Web Services:

- AWS Lambda:

  AWS Lambda is a serverless, event-driven computing service. We are using AWS Lambda to run our application (backend service) virtually without provisioning or managing servers. The file will be downloaded from the Input S3 bucket by our Lambda function, which will then process it to recognize the face. In order to obtain more details about the student, it will next interact with DynamoDB (our database), build a CSV file, and upload it to the Output S3 bucket.

- Amazon DynamoDB:

  Amazon DynamoDB is a fully managed, serverless, key-value NoSQL database. It is designed to run high-performance applications at any scale. We are using it to store academic information about students. The schema includes 4 columns, id (primary key), name (Candidate key/global secondary index), major, and year.

- AWS S3:

  S3 is a storage service provided by AWS and is used to store videos that were provided by the workload generator. It also stores the resulting CSV files from DynamoDB after facial recognition. S3 is providing persistence for our videos and results. We are using 2 buckets, one for input .mp4 videos and one for storing the output CSV files as results from facial recognition.

- Amazon ECR:

Amazon Elastic Container Registry (Amazon ECR) is an AWS-managed container image registry service. It is highly secure, scalable, and reliable. ECR supports private repositories and permissions can be assigned based on AWS IAM. We chose ECR to upload our docker image rather than DockerHub (another available option) due to the following reasons:

- ➢ better access control based on IAM user,
- ➢ provides in-depth registry usage and,
- ➢ having tight integration with Docker CLI.

Architecture Design:

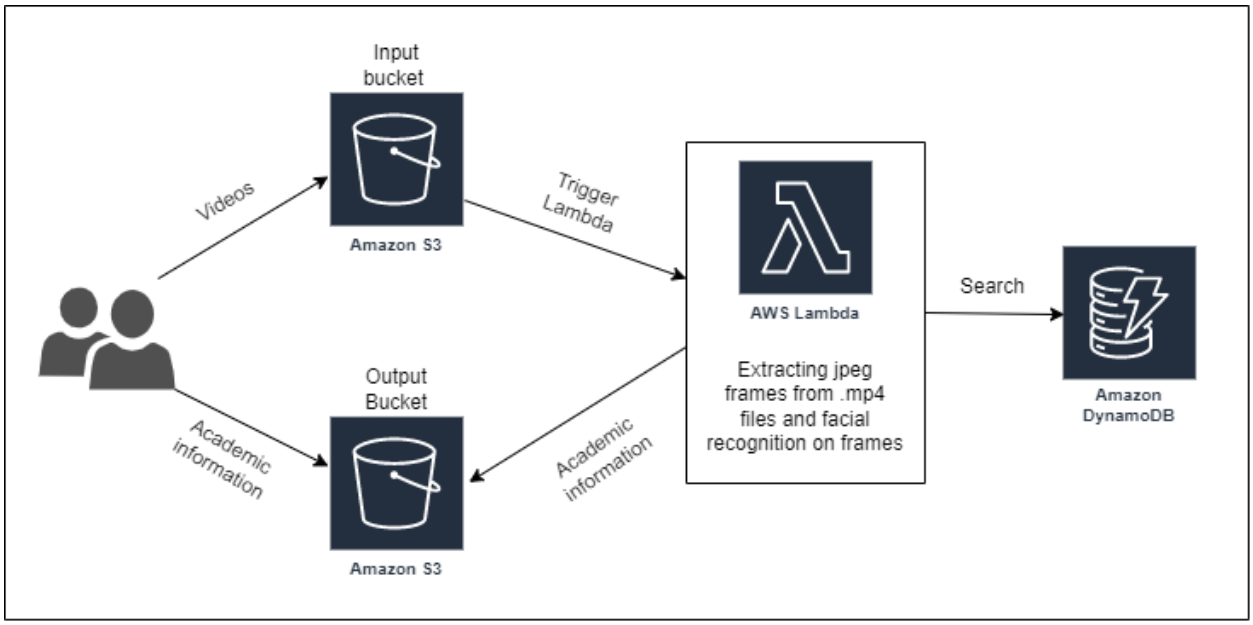

Fig.1 Architecture diagram

The Architecture diagram as seen above consists of AWS S3 buckets, AWS Lambda, and Amazon DynamoDB. The application is deployed on a Lambda function. The workload generator uploads the video files to the input S3 bucket.

The Lambda function is configured with a trigger event for its invocation. The event is configured on the input S3 bucket for the "object creation" event. For instance, when a video is uploaded to the input S3 bucket, the Lambda function is invoked for that input video. At the same time, the Lambda function is handling the autoscaling based on a number of invocations whenever objects are created in the S3 bucket. The workload generator will make concurrent requests to our application. The Lambda function will automatically scale up the instances required to process the requests.

For a trigger event, AWS sends the *event* and *context* object to the handler function in Lambda. These objects contain information about the triggering event and function invocation, function

configuration, etc. respectively. Our handler script uses the event object to get information about the input video to download from the input S3 bucket.

After the Lambda function is triggered and the input video is downloaded from the S3 bucket using the event information, the multimedia framework, *ffmeg* is used to extract the frames from the video. Then the script loops through the extracted frames, and upon detecting a face with known encoding the script exits. The facial recognition result is used to fetch further details from Amazon DynamoDB. And these results are then written to a CSV file which is uploaded to the output S3 bucket.

2.2 Autoscaling

Concurrency is the number of in-flight requests our AWS Lambda function handles at the same time. For each concurrent request, Lambda will provide a distinct instance (container) of our docker image (execution environment) that was uploaded to the ECR. Lambda will automatically scale the number of execution environments as our function receives more requests up until the account's concurrency limit is reached.

The default concurrency limit for all functions in a region set by Lambda is 1,000. As our maximum limit is 100 requests at a time, we are using the in-built scaling provided by AWS Lambda.

2.3 Member Tasks

We have added the Individual Project contributions under section 5.

3. Testing and evaluation

- The steps mentioned below were followed in order to test our application:
    - The application was tested with the 100 videos that were provided as a part of the assignment.
    - We then verified if the S3 input bucket is receiving the videos in the desired format.
    - We also monitored the Cloud Watch logs for the AWS Lambda function.
    - And verified the CSV files in the output bucket.
- The following are the results of testing and evaluation:
    - The results generated by the classification are stored in a .csv file in the S3 for persistent storage.

4. Code

4.1 Files

- Dockerfile
  - To build the docker image to encompass all the required source code files such as handler.py, encoding, and package dependency files such as requirements.txt for the application.
- encoding
  - This file contains the encoding for the known faces.

- entry.sh
  - This shell script contains the commands for the *ENTRYPOINT* of the dockerized application. This script runs every time a container of the application is created.
- handler.py
  - This python script is the main function of the AWS Lambda service. And this is called when the Lambda function is triggered.
  - This file also contains the face recognition application logic.
- mapping
  - This file contains the input video name and the corresponding output.
- requirements.txt
  - The dependency file containing the name of all the required packages for the application to run.
- student_data.json
  - This file contains the list of all JSON objects, where each object corresponds to a single record in the database.

4.2 Installation and Running

- The application is to be dockerized for which the *Docker* component is needed on the system. And *AWS CLI* is also configured to connect with the ECR repository.
- Following are the commands that were executed to set up on the system:

  >> aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin <registry_url>

>> docker build -t <image_name>:<version> .

>> docker tag <image_name>:<version> <registry_url>/<image_name>:<version>

>> docker push <registry_url>/<image_name>:<version>

5. Individual contributions

5.1 Muskan Mehta (1225444701)

DESIGN:

The workflow of the application can be described as a two-step process.

- The first step is to extract frames from the input video (.mp4) file and stores them in a temporary folder(/tmp folder) in the AWS Lambda instance.
- The next step is to loop through the frames extracted one by one. We fetch the first frame (image) which contains a known face from the encoding. Once the frame is identified and recognized, it is mapped with the respective details from DynamoDB and the result is stored as CSV in the output S3 bucket.

IMPLEMENTATION:

The following steps were taken as part of the implementation:

- In the handler.py script I worked on the function fetch_db_item. It has two arguments: name and video_name.
- After being called by the face_handler function, with the recognized face name (the first argument: name), the above function scans and fetches all the academic information from the DynamoDB table.
- This function also contains the code to store the results from DynamoDB into a CSV file. It will then use the second argument, video_name to name the CSV, and upload that CSV file to the output S3 bucket.
- I also worked on the creation of a DynamoDB table with the *id* column configured as the Primary key for the table. The *name* column is configured as the global secondary index/candidate key to make scanning the database easier.
- For the DynamoDB table data loading, I modified the given JSON data file so that AWS can consume it.
- I worked on the creation of the input and output S3 buckets to store the input videos and output CSV files respectively.

RESULTS:

- The input and output buckets received the data as expected.
- A major takeaway from the project is learning about docker images (containers) and how the image can be built using the Dockerfile.
- I learned about the various commands used for docker files and about the various resources related to Dockerfiles.

5.2 Sannidhya Pathania (1225329976)

DESIGN:

- The main objective of the project was to make an elastic application that can automatically scale in and scale out on demand. Since, AWS Lambda is serverless, and lightweight for the developers, it was one of the top choices for fulfilling the requirements. I first went through the autoscaling provided by AWS Lambda. As the default limit for requests was 1000 at a time, I decided to use AWS Lambda's autoscaling for our project.
- At the same time, the Lambda function was provisioning with other requirements. AWS Lambda can autoscale over the deployed docker images provided by ECR. Thus, Lambda was a good choice for interacting with container images.

IMPLEMENTATION:

The following steps were taken as part of the implementation:

- In the handler.py script, I developed the function face_handler.
- This function is the trigger point for the application.
- The function is executed for each object that is created/uploaded to the input S3 bucket.
- The function takes in two arguments, event, and context.
- The event contains the information required for the particular lambda instance to determine which S3 object it has to download.
- The context is not used in our application, but since this is the default argument passed by AWS, I included it as an argument.
- The function also loops through the extracted frames and detects the frame with known face encodings. After recognizing the face, it calls the fetch_db_item to get the academic information for a particular student from DynamoDB.
- I also worked on the creation of an ECR repository to store the docker images.
- The repository was configured to be private as it was a backend application and only required communication with services within AWS.

RESULTS:

- The application was autoscaling as expected. The function face_handler was able to get the event information and process the request further and called the fetch_db_item accordingly.
- I learned about how Lambda service works as serverless to deploy an application and about the limit on Lambda's auto scalability.
- I learned about the ECR repository provided by AWS, which is highly secure and reliable.

5.3 Vaishnavi Amirapu (*1225461211*)

DESIGN:

- For this project, the same process was performed as in Project 1. The concept of least privilege access is used where only the permissions required by the users are given.
- The new AWS services used in this project are AWS Lambda, AWS ECR, and AWS DynamoDB for read and write operations. The policies were added to the existing IAM group.
- Since the application is to be dockerized and pushed to the ECR repository, the local systems of the team members were also set up with the components - Docker and AWS CLI.

IMPLEMENTATION:

As a part of the implementation steps the following tasks were performed:

- In the handler.py script, I worked on the frames_generation and frames_deletion functions.
- In the first function, I used the multimedia framework ffmeg to extract the frames from the input video and store the frames in the temporary folder /tmp/.
- In the second function, I worked on clearing the temporary folder to delete all the frames generated, and the downloaded input video.
- I also worked on dockerizing the application and pushing it to the ECR repository. And configured the Lambda function to deploy the pushed docker image.
- A trigger event is also configured for the function on the input S3 bucket whenever a video is uploaded to it.
- The function is configured on the default architecture provided by AWS which is x86_64. The timeout value is set to 5 minutes. And the ephemeral storage is configured to 4GB in order to store the input video and its extracted frames.
- I also ensured that every time a new version of the docker image is built and pushed to ECR, then it is also updated on the AWS Lambda function.

RESULTS:

- The application functions worked as expected in generating the frames for the input video and clearing the frames upon task completion.
- One of the key takeaways of this project was to learn about the service, Amazon ECR, which is used to store docker images. Also learned on how to authenticate with this private registry and push the docker images.
- I also learned of how to use the ffmeg framework and the facial recognition library to detect the face and compare it with the known encodings.