

CSE 546 — Project2 Report

<i>Ashish Kumar Rambhatla</i>	<i>1215350552</i>
<i>Chaitanya Prakash Poluri</i>	<i>1223158937</i>
<i>Reethu C Vattikunta</i>	<i>1222619619</i>

1. Problem statement

Our project's main motto is to build a serverless distributed cloud application that provides the Image recognition services by using AWS Lambda, a FaaS (Function as a Service) resource provided by Amazon Web Services (AWS). The video feed from the raspberry pi camera is streamed to AWS cloud infrastructure for facial recognition. The serverless applications' motto is to identify the person in front of the camera of a Raspberry Pi unit. To provide real-time performance, the frames from this feed should be extracted and passed to lambda functions for performing facial recognition using a custom trained model. The recognised person's academic details are fetched from AWS DynamoDB, which is pre-configured with student data. The academic information obtained from the dynamoDb should be returned to the raspberry pi. As the main purpose of using cloud infrastructure is to provide real-time performance, the application should be optimized to obtain minimal latency while providing reliable accuracy. This project could play a vital role in many industrial and consumer applications such as monitoring manufacturing pipeline units, smart doorbells, security cameras, etc. The AWS services used in this project are AWS Lambda, AWS DynamoDB, AWS Identity access management, AWS API Gateway, AWS S3.

2. Design and implementation

2.1 Architecture

High Level Architecture:

The videos are captured with the Raspberry Pi's connected camera. The captured video feed is streamed to the S3 bucket for ease of access for all the AWS services. The AWS lambda function which performs the facial recognition functionality is integrated with an AWS API Gateway. This API gateway acts as a frontend for all the AWS processing. In our implementation, the raspberry pi first uploads the extracted frame to an S3 bucket and then makes a GET request to the API gateway to fetch the recognition results. To provide flexibility to change the buckets, the GET request is configured to have *BucketName* and *ImageName* as query parameters. These parameters are passed as a JSON event to the lambda function through "lambda proxy integration" and serve as an input to the lambda function. Once a user makes a GET request with appropriate parameters, the Lambda function is triggered and will do the following functions.

1. Fetch the query image from the S3 bucket.
2. Perform facial recognition on the query image.
3. Use the prediction label as the partition key to search the dynamoDb table for getting the academic details.
4. Build a http response JSON object with the academic details and return them back to the raspberry pi.

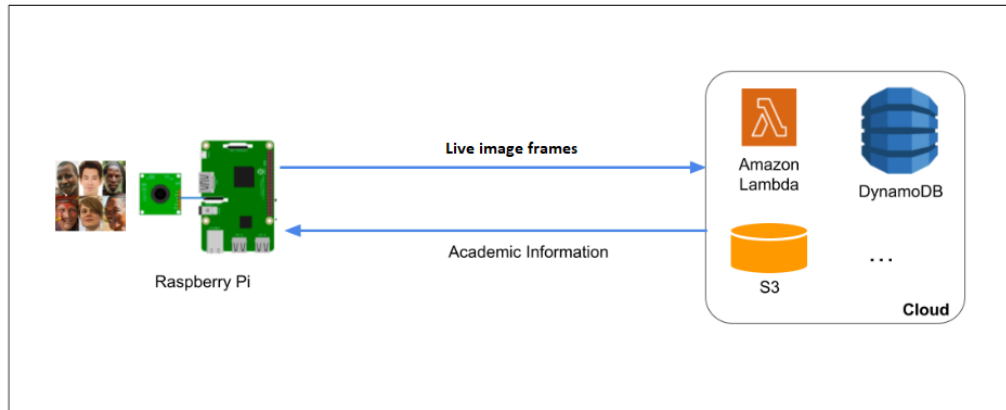


Figure1 : Provided architecture diagram

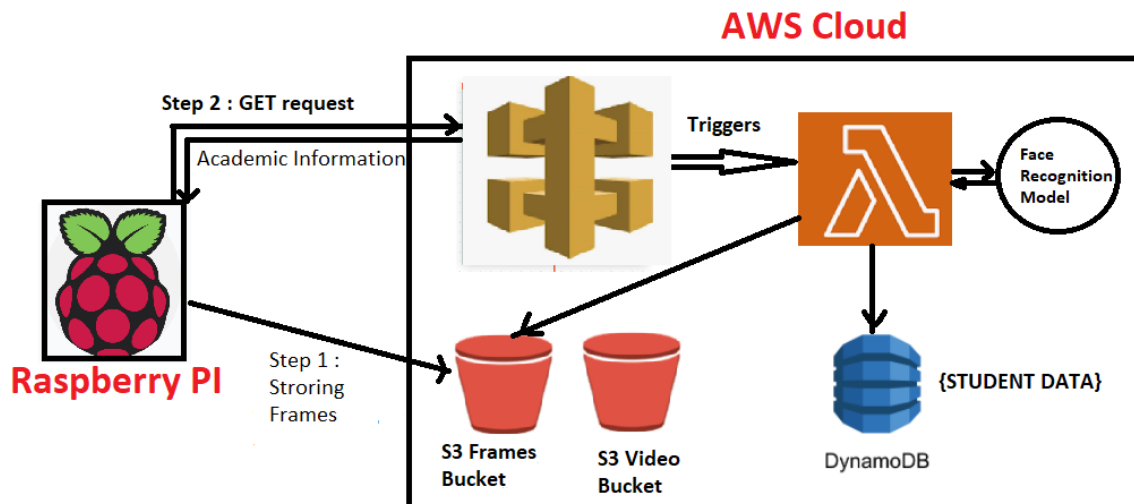


Figure 2 : Implementation of our Project Architecture

Implementation of each component in detail:

→ Components that are being used in the Project:

- ◆ The Raspberry Pi
- ◆ Customized Face Recognition Model
- ◆ AWS Lambda

- ◆ AWS S3
- ◆ DynamoDB
- ◆ AWS API Gateway

The main IoT device in this project is the Raspberry Pi. The AWS components that are used are AWS Lambda, AWS S3, DynamoDB and AWS API Gateway. There are other components that are used like Dockerfile, AWS Elastic Container Registry and AWS CloudWatch. For example, AWS CloudWatch is used for monitoring lambda metrics such as invocation counts and debugging the lambda executions.

AWS Lambda:

AWS Lambda enables users to develop business logic of an application as a function in a runtime environment. AWS provisions the servers and configures the environments in the backend to support these run-time environments. There are multiple ways to configure the lambda environments such as 1) writing a function from scratch, 2) using a container image stored in ECR, 3) Using SAM local development and cloud front infrastructure. In this project, we are using a container image to configure and deploy the lambda function. In our project, all the business logic is being done in the Lambda Function. The first step in the Lambda functions is to fetch the images from the S3 bucket. Then perform face recognition on the downloaded images and use the prediction label to search dynamoDB for academic details. These details are then sent to the raspberry pi as output.

AWS S3 Service

Amazon Simple Storage Service is a storage service offered through a web interface for objects (image, videos, text, etc.) storage. We are using two S3 buckets here for persistent storage and ease of availability of data across AWS services. The first use case of the S3 bucket is the persistent storage of the extracted frames. This S3 bucket also acts as input for the lambda function by hosting the extracted frames. The second use case of the S3 bucket is for storing the whole video in the S3 bucket. Lambda is triggered using the API gateway using the Image name and bucketname. Lambda downloads the frames from the “g45-frame-extraction-bucket” and runs them against the face evaluation model. We are also streaming the video all the time while extracting the frames and finally storing the video in “g45-vid-input-bucket”.

AWS API Gateway

AWS API Gateway comes along with RestFul API management functionality. It is most recommended to use when we are designing large or complex micro service-based applications with multiple clients. These provide a really faster way to interact with the backend services. It can serve as a single entry point for various clients providing specific APIs for each of them. It also helps in improving security. We are using an API service in our project to invoke the lambda function. A ‘*facerecognition*’ resource under which a GET method is defined. The endpoint URL of the ‘*facerecognition*’ is used to query the face recognition results by passing the imageName and bucketName where it is stored as the query parameters. Lambda executes its part and provides a recognition of the person in the frames and extracts the academic details of the

person from the DynamoDB. This is returned to the raspberry pi as a response of the RestFul request made initially by raspberry pi. The result in the raspberry pi will be a string with name, major, and year of the person.

DynamoDB:

Amazon DynamoDB is an unique NoSQL database platform that is fully managed and supports document data types and key-value cloud services. In our Project, we are using Dynamo DB when the name of the student is obtained from eval_script.py(Face recognition model). With name as the partition key, we fetched the academic details with the Dynamo DB query. We then return the obtained output.

Elastic Container Registry:

The elastic container registry is used as a persistent storage registry for the docker images. These docker images have all the required code and files to configure and deploy the lambda function. We have used “docker-desktop” to build and push the docker images to the ECR registry.

Dockerfile:

The DockerFile is like a CMakeList file which has all the configuration required to build the docker image. In this file, the lambda runtime layer is first taken as a base and other required layers such as the aws lambda runtime interface are added on top of it. This file is also used to create the directory structure required for lambda execution by copying them from the local system to the docker image. At the end, the entrypoint and lambda handler functions are defined for the lambda to configure its execution environment.

Architecture Implementation:

The live videos are captured by the camera on the Raspberry Pi, and the live image frames are extracted with the python script on the Pi. When the frames are extracted in the Pi they are sent to the S3 Bucket “g45-frame-extraction-buckert”. Post upload we raise a GET request to the AWS API Gateway every 0.5 seconds. This GET request contains the frame_name and the bucket name for the lambda function to use. This API will trigger the lambda function using an event in a JSON format. These will take the frame name and bucket name and use them to pull the image from the storage and run the evaluation of the person in the image against a deep learning model. The prediction results from the previous step are used as a partition key to extract the academic details of the person from the DynamoDB. The details of the people involved in the project are initially stored in DynamoDB in NoSql format. These DB results are returned from the lambda function which will be returned as a http response to the

raspberry pi. The whole video that is streamed during the duration is sent to another S3 bucket named "g45-vid-input-bucket".

2.2 Concurrency and Latency

Concurrency and latency are the vital parts of our project. With them in consideration, we made several optimization decisions and architecture changes to achieve concurrency and to improve the latency. We implemented multi-threading in the raspberry pi to send http requests to lambda asynchronously while also writing every frame to a video file locally. For this purpose, we used a threading module in python to spawn multiple threads to perform both actions independent of each other.

On the other hand, the lambda function is written in such a way that it can process frames individually which allows us to invoke multiple lambda functions concurrently based on the demand. The load-balancing of the lambda functions is handled by the AWS in the backend based on the density of incoming triggers.

3. Testing and evaluation

Testing is one of the most important steps in the development of this application, we have tested it from end to end and optimized latency, performance and storage factors. We initially implemented the whole frame extraction thing in the AWS lambda but later to increase the response time and decrease the throughput we decided to follow the frame extraction from the pi. This has really helped us improve our latency from 7s to 2.5 s. The same happened when we used SQS initially for the messages to send from lambda to pi, there has been a need for polling of messages. These really affected our latency. But we learned a lot of ways of implementing the same application in different ways.

Another important size is Docker size reduction, initially due to deep learning libraries and the training and validation data the container data size is as big as 3.5 gb. Later we removed the unnecessary files from the docker container and used this image. This finally led to reduced size in the Elastic Container Registry.

g45-vid-input-bucket [Info](#)

[Objects](#) | [Properties](#) | [Permissions](#) | [Metrics](#) | [Management](#) | [Access Points](#)

Objects (3)

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

Copy S3 URI

Copy URL

Download

Open

Delete

Actions ▼

Create folder Upload

< 1 > ⚙️

<input type="checkbox"/>	Name ▲	Type ▼	Last modified ▼	Size ▼	Storage class ▼
<input type="checkbox"/>	23_04_2022_17_17_01.mp4	mp4	April 23, 2022, 17:17:08 (UTC-07:00)	8.7 MB	Standard
<input type="checkbox"/>	23_04_2022_17_22_41.mp4	mp4	April 23, 2022, 17:22:47 (UTC-07:00)	8.6 MB	Standard
<input type="checkbox"/>	23_04_2022_17_27_45.mp4	mp4	April 23, 2022, 17:27:51 (UTC-07:00)	8.7 MB	Standard

The above g45-vid-input-bucket shows the video bucket representing the videos taken for all the duration. These videos are all taken at various testing phases.

g45-frame-extraction-bucket [Info](#)

[Objects](#) | [Properties](#) | [Permissions](#) | [Metrics](#) | [Management](#) | [Access Points](#)

Objects (600)

Objects are the fundamental entities stored in Amazon S3. You can use [Amazon S3 inventory](#) to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)

Copy S3 URI

Copy URL

Download

Open

Delete

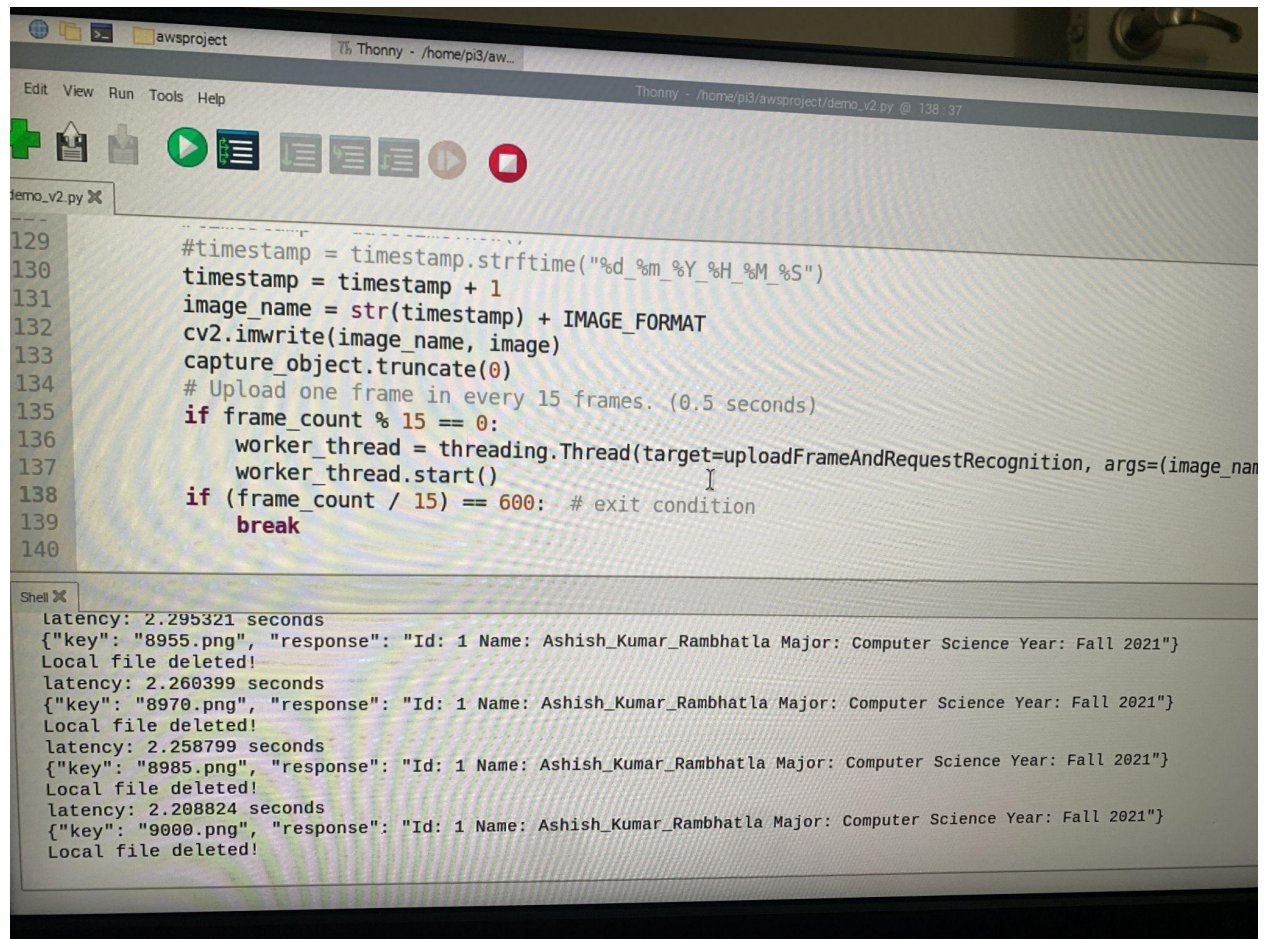
Actions ▼

Create folder Upload

< 1 2 > ⚙️

<input type="checkbox"/>	Name ▲	Type ▼	Last modified ▼	Size ▼	Storage class ▼
<input type="checkbox"/>	1005.png	png	May 6, 2022, 22:42:52 (UTC-07:00)	34.5 KB	Standard
<input type="checkbox"/>	1020.png	png	May 6, 2022, 22:42:53 (UTC-07:00)	34.8 KB	Standard
<input type="checkbox"/>	1035.png	png	May 6, 2022, 22:42:54 (UTC-07:00)	34.4 KB	Standard
<input type="checkbox"/>	105.png	png	May 6, 2022, 22:42:18 (UTC-07:00)	31.7 KB	Standard

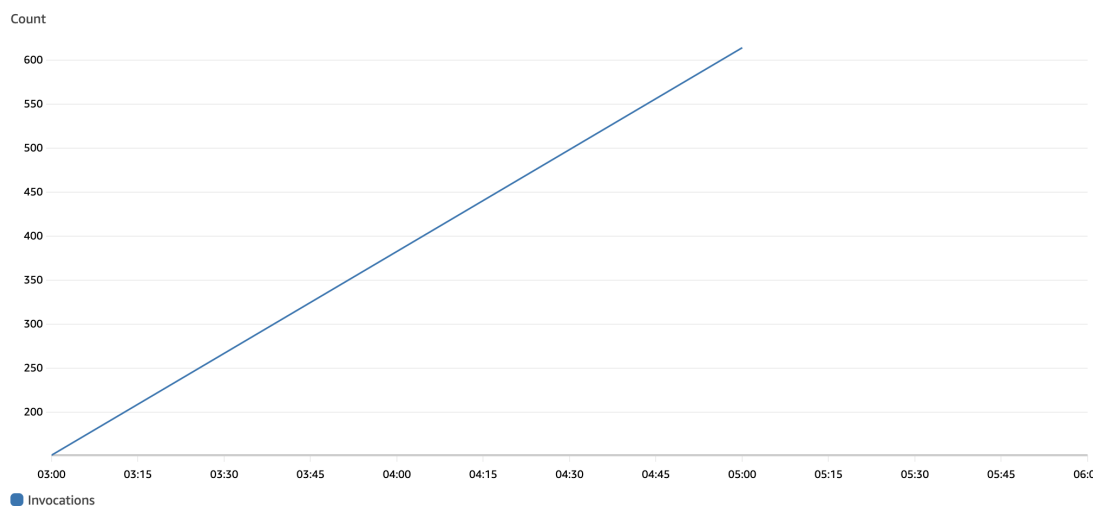
The frame extraction bucket represents the pictures frames extracted from pi.



```
demo_v2.py
129 #timestamp = timestamp.strftime("%d_%m_%Y_%H_%M_%S")
130 timestamp = timestamp + 1
131 image_name = str(timestamp) + IMAGE_FORMAT
132 cv2.imwrite(image_name, image)
133 capture_object.truncate(0)
134 # Upload one frame in every 15 frames. (0.5 seconds)
135 if frame_count % 15 == 0:
136     worker_thread = threading.Thread(target=uploadFrameAndRequestRecognition, args=(image_name,))
137     worker_thread.start()
138 if (frame_count / 15) == 600: # exit condition
139     break
140
```

```
Shell
latency: 2.295321 seconds
{"key": "8955.png", "response": "Id: 1 Name: Ashish_Kumar_Rambhatla Major: Computer Science Year: Fall 2021"}
Local file deleted!
latency: 2.260399 seconds
{"key": "8970.png", "response": "Id: 1 Name: Ashish_Kumar_Rambhatla Major: Computer Science Year: Fall 2021"}
Local file deleted!
latency: 2.258799 seconds
{"key": "8985.png", "response": "Id: 1 Name: Ashish_Kumar_Rambhatla Major: Computer Science Year: Fall 2021"}
Local file deleted!
latency: 2.208824 seconds
{"key": "9000.png", "response": "Id: 1 Name: Ashish_Kumar_Rambhatla Major: Computer Science Year: Fall 2021"}
Local file deleted!
```

caption: picture showing the processing of 9000 frames (30fps * 300 seconds) and the avg latency as 2.23 seconds



caption: number of lambda invocations is equal to 600 frames.

We have tested our code by running the raspberry pi for the complete 5 minutes duration and captured latencies at different components of the architecture such as face recognition, lambda execution to analyze the delay involved in each component. This knowledge is used to optimize the architecture and different components involved in it. Overall we 3 architectures and finally settled with the architecture that gave high performance.

4. Code

Our project has the following 3 Parts:

- Raspberry Pi
 - [demo.py](#)
 - [recordVideos.py](#)
- Face Recognition
 - [app.py](#)
 - [eval_face_recognition.py](#)
 - [Dockerfile](#)
- Dynamo DB
 - [dynamodbUpload.py](#)
 - [student_data.json](#)

Raspberry Pi Code:

demo.py

This file has mainly three functionalities, 1) record videos, 2) upload to the S3 bucket, 3) Send a GET request to API gateway for face recognition results.

The main function of the demo.py starts with the recordVideos() method. This method first initializes the pi camera. The picamera python module is used to create a *capture* that is configured with the required camera resolution and frame rate. Then in a loop, the frames are read from the camera feed at the configured frame rate. These frames are passed to “uploadFrameAndRequestRecognition” function which first uploads the frame to the “g45-frame-extraction-bucket” and then triggers a GET request with appropriate parameters to query the face recognition results of the uploaded image from the lambda. Finally the local video file is pushed to the Video S3 bucket for persistent storage.

Lambda Code:

DockerFile:

The dockerfile provided by the professor is used as a base to configure our lambda functions. The lambda runtime layer is used as a foundation in this file, with other essential layers such as

the aws lambda runtime interface put on top. This file is also used to copy directory structures from the local system to the docker image, which is essential for lambda execution. To configure the lambda's execution environment, the entrypoint and lambda handler functions are defined at the end.

One specific change that we did to meet the lambda's read-only file system constraint is to redirect the *torch_home* directory to */tmp/* folder as that is the only writable directory in the lambda function.

Requirements.txt:

This file lists all the required python libraries required by the lambda function runtime to execute the obtain the required functionality

run_prediction.sh:

This shell script consists of the python command to run the "eval_face_recognition.py" file with a placeholder for the image name using the "\$1" symbol.

app.py:

This python file hosts the entrypoint for the lambda function to start its execution. This file has all the functions to execute the face recognition functionality. First the code starts with "lambda_handler" method. This method first reads the event data to get the query parameters such as ImageName and BucketName. With this information, the image is downloaded from the S3 bucket to the local file system in *"/tmp"* directory. Now the *"run_prediction.sh"* file is forked using the subprocess python module. Once the prediction results are received in the form of a string, we parse those results using *"parse_results"* function to find the prediction label. Later this label is used to form a key structure to search the dynamodb table to fetch the academic details. Finally a http response is constructed using *"buildResponse"* function and sent back to the raspberry pi.

eval_face_recognition.py:

This file has the classification logic, given an input image file. it uses other supplementary scripts which also need to be present in the same directory as this file.

Training:

training_face_recognition.py:

This file has the business logic to parse the dataset present in *"data/real_images/"* folder to train a custom model based on the inception_v1 architecture. It also generates the labels.json file based on the directory names present in the *"data"* folder.

PROJECT SETUP:

First, configure the Raspberry Pi by installing the operating system, flashing it on a disk, and connecting the drive to the hardware. Then connected the Raspberry Pi to the WiFi by using VNC Viewer. Created and programmed AWS S3 buckets, Lambda functions and Dockerfile. Login

into Pi, executed the demo.py file from the Project folder, and executed the python script to obtain console output at the same time. As everything is automated, the demo.py will automatically call for API Gateway and thus triggers the Lambda Function, followed by face recognition, the output fetched from the Dynamo DB is send back to the Pi.

5. Individual contributions (optional)

Reethu Chowdary Vattikunta(ASU ID - 1222619619)

AWS has been one of the top providers of cloud services for the last two decades. The indirection layer has come a long way since EC2. Now we are able to design serverless architecture for cloud applications. In this project we are trying to develop a cloud application based on AWS services such as Lambda, DynamoDB, API Gateway. I actively contributed to the team in the implementation and designing phase.

Design Phase:

I have contributed to training the machine learning part and setting up the PI. I have set up the Pi using the connection implemented by the professor. Later I used the classical ssh connection to connect with the ip address. I also used Putty and VNC client to visualize the contents in the Pi. As we have done the training twice, I have collected the data, modified and uploaded into the AWS EC2. Later also trained locally. The second part of the training involved taking pictures. After capturing videos, frames are extracted for every second of the 40 seconds video. The extracted pictures are used for training and evaluation. When I obtained an accuracy of around 60%, my teammate then integrated it into the cloud. Also contributed to writing the Report, especially for Design and implementation, and code.

Debug Phase:

I helped my friends in modifying the training logic and the multithreading logic. I performed the end to end testing and debug the whole code to optimize the solution.

5. Individual contributions (optional)

Chaitanya Prakash Poluri (ASU ID - 1223158937)

We all know that AWS is one of the top service cloud providers. In this project we are developing a serverless application for image recognition. For this application we are using AWS services such as AWS lambda, AWS S3, AWS DynamoDb, and AWS API Gateway. We are also using concepts of containers, Dockers, Raspberrypi, multithreading, and APIs. The ultimate goal of this project is to provide the academic details of the person in front of the camera in real time. This is one of the crucial applications of Edge computing. I have contributed to the team at various stages of Design, Development, and testing.

Design Phase:

I along with my teammate took appropriate measures following the architecture. My contributions to the team are in designing the video processing and image capturing logic in the raspberry pi. I have created two S3 buckets from the AWS management console for storing the images and the video of the overall time duration. I helped my teammate in designing the API gateway and when to trigger the lambda function. I have initially setup the trigger through S3 and output from the DynamoDb to pi through SQS and SNS reportedly. But later to improve the latency from end to end we choose AWS Gateway.

Implementation Phase:

I have initially contributed to the effective images for better accuracy. I connected to the cloud using ssh and pulled all the files locally through scp later gave all the code to my teammates. She trained the model locally and validated it. I used that implementation and also checked in the EC2. I have created two S3 buckets from the console, one for the images and another for the whole video. I helped my teammate in setting up the pi and later designed the video and frames logic from the pi and using the S3 client I stored the frames and videos in the respective buckets. Initially when I was trying the previous architecture I implemented the lambda trigger through S3 service and output through SQS and SNS .I also implemented the multithreading logic in the pi to send frames to improve the latency.

Testing Phase:

The whole process has not been easy with proper coordination from teammates as the physical resources are limited. We had to divide and then share the resources available. Initially we had bad latency but we tested it end to end many times to improve it. We initially set up the lambda to trigger based on S3 but later implemented it with API gateway. I tried SQS and SNS services to send the messages to the Pi initially. But after the full functional testing we settled with API Gateway. In this process we came across various architectures. The one we followed gave a best accuracy of 2.5 seconds.

Debugging Phase: I helped my teammates in modifying the training script and evaluation.py script to better suit the model for our purpose. I took help while designing the multithreading concept from a friend when the script ran into unknown errors.

5. Individual contributions (optional)

Ashish Kumar Rambhatla (ASU ID - 1215350552)

Design Phase:

I have pioneered the design and architecture phase of this project. I have initially designed the project to have a s3-triggered lambda function which then runs the recognition functionality and db query search. The results are then published on a sns topic which pushed to a FIFO queue. The raspberry pi polls this queue to get the results. This approach did work but it resulted in a huge amount of latency (~ 4-5 seconds) due to the involvement of the SNS and polling methodology on the raspberry pi side. I have then re-architected the whole project by changing the trigger to API gateway which has far lesser latency. The api is designed to return the face recognition results to a GET request accompanied by the image name as a query parameter. I have also contributed to the deployment of the lambda functions and also the dockerisation process.

Implementation Phase:

I have contributed to the development and deployment of the lambda and api gateway parts of the project. In the lambda implementation, I have designed it to parse the input trigger to fetch the image name and bucket name details. This provides the flexibility to change the input bucket at run-time without updating the lambda. Later I have implemented the other parts of the lambda to achieve the desired functionality. I have also implemented the dockerisation part of the project by configuring the dockerfile to include all the local files in the development environment, import all the python dependencies and also the aws lambda runtime interface. I have also solved the "read-only access" errors in the lambda function by redirecting the torch_home to /tmp directory.

Testing Phase:

I have taken quite an effort to test each component individually at the development phase. For example, after developing the lambda, i have configured the test events to actually test the lambda individually before integrating it with the API gateway.

After integrating lambda with api-gateway, i have tested this system using postman by querying the endpoint url with the sample query parameters.

Later I integrated the pi into the system. Due to the step-by-step testing procedure i followed throughout the development process, it became hassle-free by the time i integrated pi into the system.

Debugging Phase:

I have helped my group members in changing the training parameters such as batch size and adam optimizer values to achieve better accuracy. I also helped my teammates in the video processing pipeline and debugging techniques to ace the development of the same.