CSE 546 — Project 2 Report

Anup Kashyap (akashy15@asu.edu)
Krutarth Bhatt (kmbhatt2@asu.edu)
Sunaada Hebbar (shebbarm@asu.edu)

1. Problem statement

The goal of this project is to implement a real time face recognition system that can be used in various places for various usecases. Face recognition is a common technique that is used to identify people. It can be used for applications such as automated attendance systems, tresspasser detection or even to authenticate people and grant access to certain resources. In this project we use a raspberry pi as an edge device with a camera module attached to perform face recognition. We also offload the face recognition computation to a cloud service in a serverless manner. We make use of AWS lambda for computation and S3 and dynamodb for storing data. The final project detects faces from the raspberry pi camera and returns the information about the student.

2. Design and implementation

2.1 Architecture

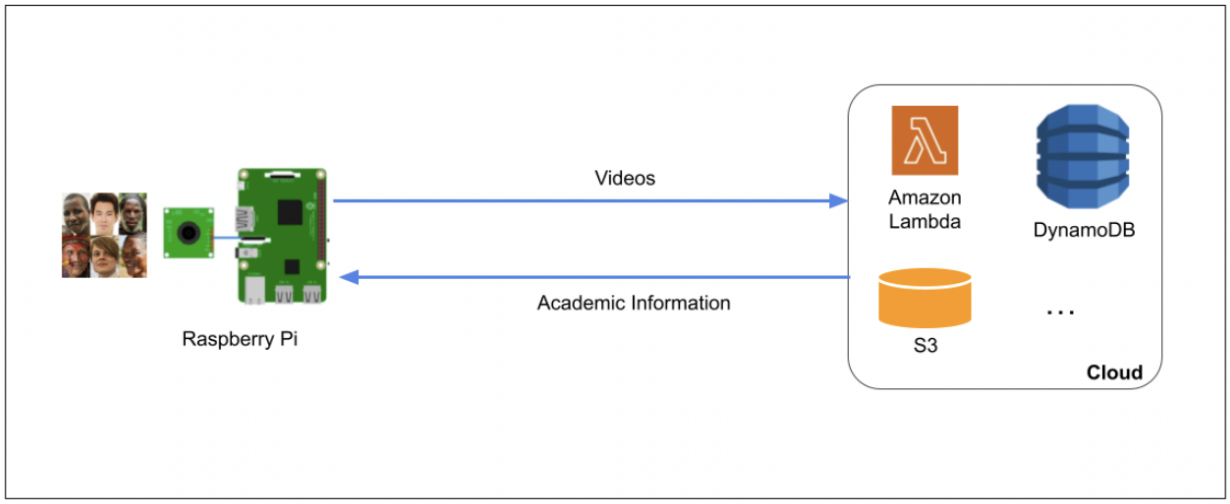

Figure 1: Block diagram of the face recognition service

The high level architecture of the face recognition service is shown in Figure1. We make use of a serverless architecture. AWS Lambda is a function-as-a-service (FaaS) offering of AWS where we define a function without having to worry about the underlying hardware and the operating system. We also use DynamoDb which is a nosql database offering of AWS. We use dynamoDb to store student information. S3 is used to maintain a record of the videos that were captured by the face recognition service. The

overall flow of the process and the interactions between components of the system is shown in Figure 2.

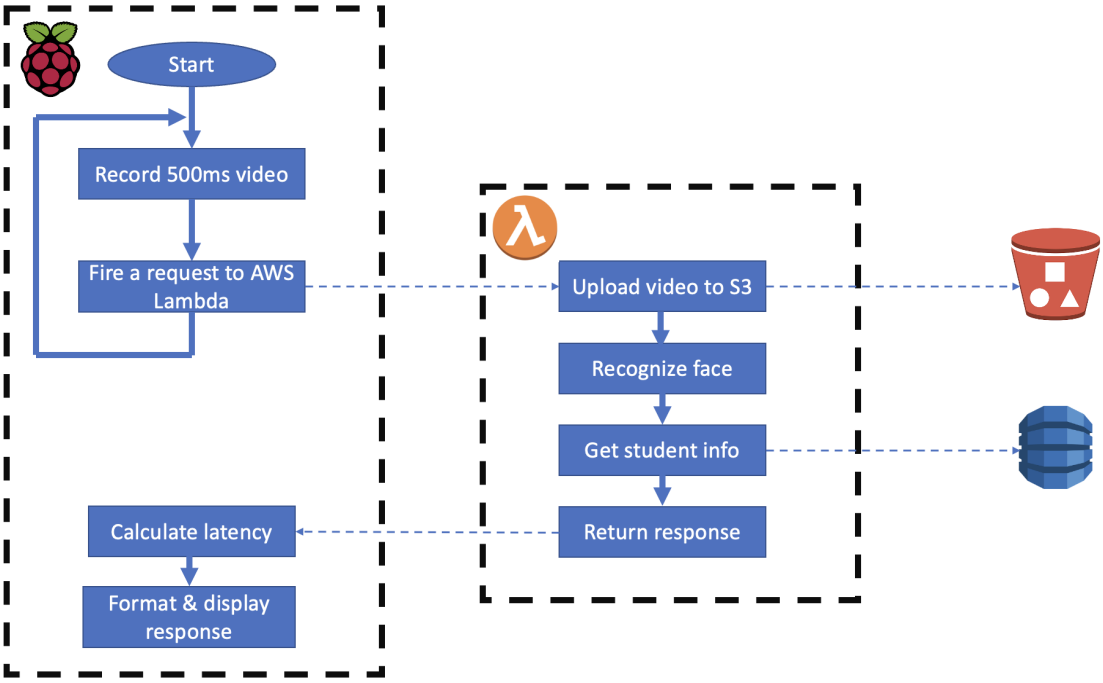

Figure 2: Overall process flow of face recognition service

2.2 Concurrency and Latency

The design implemented uses multiple threads to run different processes of video collection and concurrently running inference on the videos being collected. Figure 3 depicts how concurrency is achieved by the software running on raspberry pi.

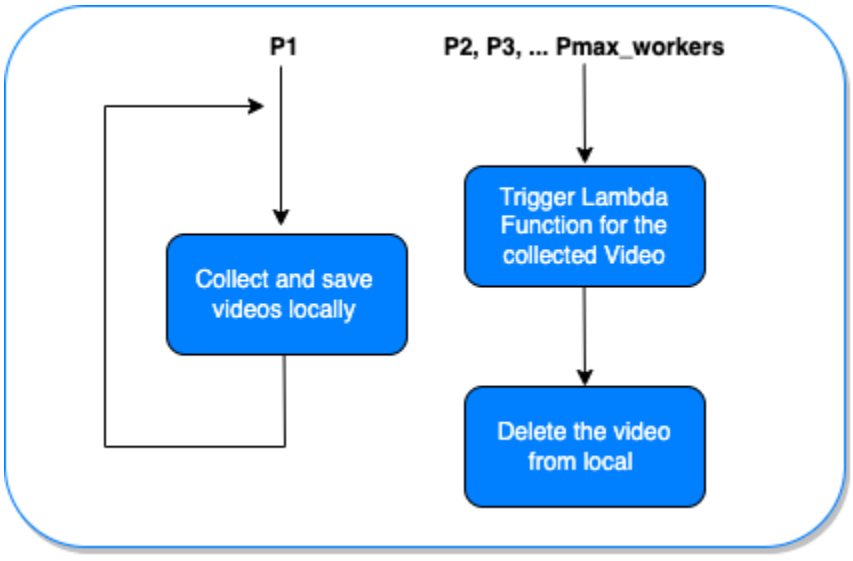

Figure 3: Concurrency of the software running on raspberry pi

There are different dedicated threads for collecting the videos and invoking lambda for obtaining results. We use the python package **ThreadPoolExecutor** to achieve concurrency. The latency in the

process is calculated as the time taken to upload and save the video to S3, Preprocess the video (Extracting Region Of Interest from the frames), running the trained face recognition model to recognize faces, and returning back the response. Our method achieves the latency of average 1.5 secs for each request.

3. Testing and evaluation

Testing the real time face recognition service was executed in multiple stages:

- Model Testing:
    - Process: In the first stage we tested the face recognition model trained with the images captured by the raspberry pi camera. We recorded videos of different team members, extracted the frames and evaluated the model with extracted frames.
    - Initial Result: We evaluated the trained model with 10 images of each team members and found that the model did not performed fairly well. We found that the accuracy of prediction was highly dependent on the quality of the frame extracted. Frames where the face orientation skewed, frames with low lighting, blurred pixels, background with lot of noise were typically predicted incorrectly with low confidence.
    - Improvement: We used OpenCV based library haarcascades to detect a human face, cropped the images so that they fit the face perfectly and added brightness & contrast adjustments.
    - Result: By preprocessing the dataset we retrined the model. Further, we evaluated the model with processed frames extracted from raw videos and found that the model's accuracy improved to 70%.
- Testing Lambda Function:
    - Process: Initially we tested the docker image locally on our machines. We created a script to invoke the lambda function running locally on the docker container with a sample video as input. Later, we cerated a Lambda function on AWS using the docker image. We evaluated the Lambda function on the AWS web portal by generating test events with base64 encoded data from the video files. We verified that the Lamda Funtion is executing correctly and within expected latency by checking the AWS Lamda usage statistics on AWS CloudWatch service.
    - Result: We found many bugs in our code by running the docker container on our local machine. We fixed those bugs and found that the Lambda Function deployed on AWS was not performing well the latency was in terms of 3s. Later, we increased the default virtual RAM allocated to the Lambda function to 2048 MB. This improved the performance of the Lambda Function and the latency decreased to less the 1s.
- End to End Testing:
    - Process: We executed the face recognition system end to end and recorded the results with video of team members captured in real time. Verified that the student information returned by the lambda function was accurate and matched the person in the video most of the time. We ensured that the video files were successfully uploaded to S3 and the videos downloaded from S3 were not corrupted. We exected end to end testing with different back ground setups and environmental conditions like different lighting, dayligh and artifitial light, different coloured backgrounds, etc,. Finally did an end to test for 5 minutes and ensured all the requirements of the project were met.
    - Result: We found that the overall steady state latency was within 2 seconds in most cases and the student was identified correctly in most cases. We also verified that the

total time of the videos uploaded to S3 was approximately 5 minutes and the there were around 5 concurrent invocations of lambda function.

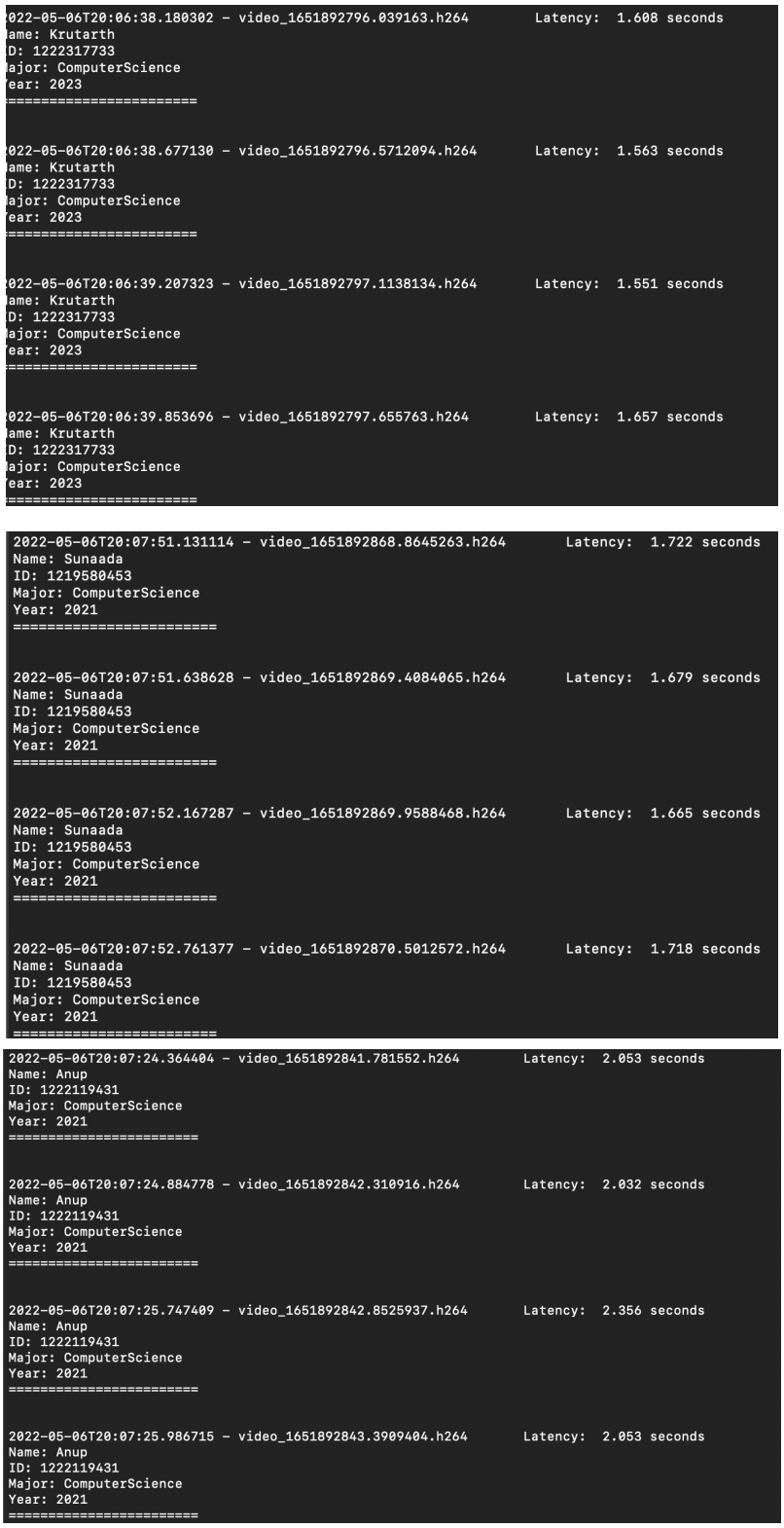

Figure 4: Screenshots showing the results obtained by the face recognition service

4. Code

```
├── README.md
├── ecs_container                          →  Lambda Container
│   ├── Dockerfile                         →  Dockerfile for build
│   ├── build_custom_model.py              →  Build Model Function
│   ├── checkpoint                         →  Model Checkpoint
│   │   ├── labels.json                    →  File with class labels
│   │   └── model_vggface2_best.pth        →  Trained model
│   ├── data/                              →  The Data Stored for Train/Test
│   ├── dockerBuild                        →  Commands to build and push image
│   ├── entry.sh                           →  Starting point in the docker instance
│   ├── eval_face_recognition.py           →  Evaluation python script
│   ├── haarcascade_frontalface_default.xml →  Data file for face detection
│   ├── handler.py                         →  handler function for AWS Lambda
│   ├── models                             →  Face recognition dependencies
│   ├── requirements.txt                   →  Python libraries and packages list
│   └── train_face_recognition.py          →  Training script
├── raspberry_driver                       →  The code Running on R Pi
│   ├── main.py                            →  The main file R Pi
│   ├── pi_camera_wrapper.py               →  Wrapper for Camera Module
│   └── results.txt                        →  File for writing results.
├── preprocessing-for-facerecognition      →  Preprocess dataset
│   ├── haarcascades
│   │   └── haarcascade_frontalface_default.xml →  Data file for face detection
│   ├── input-images                       →  Folder for input images
│   │   └── placeholder
│   ├── output-images                      →  Folder for output images
│   │   └── placeholder
│   ├── process_images.py                  →  Detect human face and improve image quality
│   └── README.md
```

The source code is available on the git hub repositories:

- https://github.com/shebbar27/cloud-computing-paas
- https://github.com/shebbar27/preprocessing-for-facerecognition

Individual Contribution (Krutarth Bhatt)

- I**mplemented Multithreading for Video Collection and Inference:**
    - Introduced the multithreading and ensured that invoking the lamda function does not block capturing video. This ensured the videos were captured in real time, the overall latency of the system was within the accepted limits and more than 3 concurrent calls were made to the lamda function.
    - Kept a single thread for collecting the videos. This thread collectes video continuously and each recorded video is of duration 0.5 seconds.
    - Used a Threadpool to invoke lambda function calls and execute them on a background thread.
- **Dataset Creation:**
    - Developed a python script to access the raspberry pi camera module and capture pictures of the team members.
    - Preprocessed the images for feeding into the model.
    - Used OpenCV based library haarcascades to detect bounding boxes of faces and cropped the region of interest and rsized the image to 160x160 pixels.
    - Applied preprocessing techniques to improve the image quality like brightness and contrast adjustments so as to make the face more clear before feeding it into the model.
    - Divided the dataset into training and testing sets for training the model and testing it on the unseen set.
    - Created a DynamoDB on AWS and created the table for the student information.
- **Model Training and Benchmarking:**
    - Trained and finetuned the face recognition model.
    - Trained the model from the collected and preprocessed datasets.
    - Tested the trained face recognition model on the collected and separated testing set.
    - Tested the model on the frames extracted from the videos using ffmpeg.
- **Troubleshooting and Documentation :**
    - Documented the working of every code in the final code base.
    - Created Diagrams and flowcharts for better understanding the working of the project and various parts of the project like (Multithreading)
    - Solved some error coming up while building the docker image.

Individual Contribution (Anup Kashyap)

I worked mainly on building a container and creating the handler function for lambda component of the project. This included the following tasks

- Writing the handler function that runs when the lambda function is invoked.
- Creating a docker image with the handler function and all the required dependencies.
- Pushing the created docker imge to AWS Elastic Container service
- Creating a lambda function on AWS with the created docker image.

Brief summary:

This entire part of the project was built using python. For the lambda function, we used the python runtime. The handler function took encoded video as the input. The video was first decoded and saved locally in the /tmp directory and then uploaded to S3. Frames were then extracted using ffmpeg library and stored into a local directory. From the directory, we go over each file and then look for faces in the frame. If a face is detected, face recognition evaluation script is triggered and based on the result, a DynamoDB call is made to get the student information. This data is retired back as the function response. This file along with all the depenncies including the trained model was copied to the docker image. And using the lambda runtime library, we map this function to be called when it is triggered. After the docker image was created, it was tested locally and once satisfied it was pushed to the ECS repository. A lambda function was created from this docker image. It was then tested by directly invoking the function on lambda. Once that was working as expected, an end to end test was run where the videos were sent from the pi's camera module.

Challenges faced:

One of the challenges that we faces was testing the lamda function that was created on AWS. It was hard to find out of the function was invoked properly in the container. Therefore, we started using the testing feature available on AWS lambda itself where we can create a test event and trigger the lambda with the created event. We created a test event mimicking our actual input by using the encoded video bytes. With this we were able to unit test the lambda function easily. Another issue that we faced was that the lambda function was timing out as it was crossing the default timeout of 3 seconds. We could get around this issue by setting the environment variable on lambda for torch home to /tmp which has write access. This solved the issue and the function was taking less than 2 seconds to run. Another minor issue was faced where the lambda function was failing because of architecture mismatch. The docker was built on M1 mac (ARM machine) and was deployed on an x86 platform. This was rectified by creating the function with ARM platform on AWS.

Individual Contribution (Sunaada Hebbar)

I have worked on the following high level tasks for the implementation of this project:

- setting up raspberry pi which includes installing OS, enabling camera module and testing the camera, establishing serial communication between raspberry pi using USB serial console and laptop and wireless connectivity
- training the face recognition model using the dataset and test the model accuracy using test images captured by the raspberry pi
- formatting the output returned by lamda funtion into the desired format for displaying it on the raspberry pi console
- testing, evaluation and verifying the performance of face recognition service including testing of different components in isolation and end to end testing of the face recognition service

Breif Summary:

I read the documentation of raspberry pi and used Raspberry Pi Imager to flash OS into the SD card provided and set up hardware. I made necessary hardware connections and set up serial communication between raspberry pi and laptop following the instructions provided. Further, I eastablised SSH connection to raspberry pi from laptop so that it was easy for the team to login to the console of raspberry pi without requiring any display or VNC server. I developed a python wrapper class which intializes the PiCamera object and exposes the api to capture video with a specified duration and specified storage location. I tested the camera module by capturing videos with custom resolution, frame rate and format. I used the dataset collected by my teammate to run the training script using the provided source code. I observed that the accuray was very low around 60% and the loss was not decreasing with each epoch. Hence, we created a new dataset with processed images. Using the new database the training accuray increased to 90% and the loss function was converging with every epoch. Furthermore, I analysed the testing and validation dataset and noted down bad images in the dataset, for example repeated images, bad lighting, blurred images, etc. Based on this feedback we created a final dataset. The final dataset achieved 95% accuracy and we used the generated model file in our docker container image. I tested the docker image buy running the docker container on my local machine. Developed python scripts which invokes the lambda function with a sample video data and verify the functionality of the container image on the local machine. Finally, I did the end to end testing of the face recognition service and recorded all the results. I analyzed the resutls and identified performace issues.

Issues Faced During Implementation:

Initially the latency was very high since we did not modify the default virtual RAM allocated to the lambda function on the AWS portal. Increasing the virtual RAM to 1024 MB decreased the latency.

In the lamda function we returned the result as json payload and converted it to dictionary to obtain the necessary results for printing on console. The format of the result had quotes the were escaped which cused issues while the parsing the result into dictionary. We fixed it by removing the escape characters.