CSE 546: Cloud computing
Project 2 Report

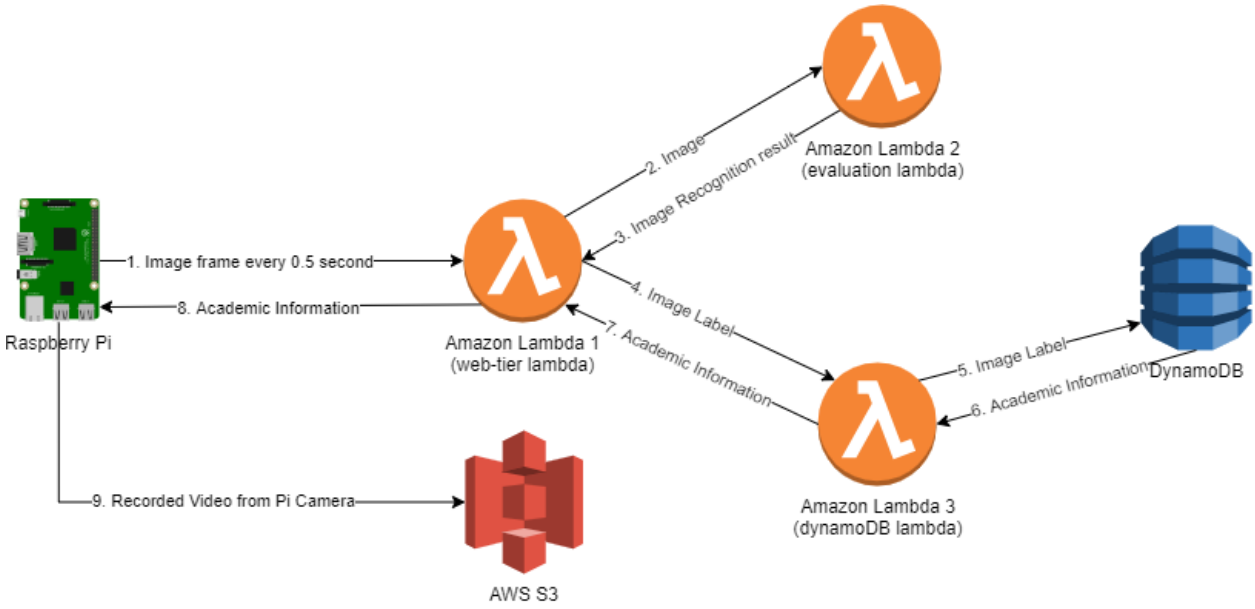
Onkar Pandit - 1222405119
Tapan Rajnikant Modi - 1222325026
Shreemad Sanskarbhai Patel - 1222713687

1. Problem Statement

In the second project, the requirement was to build a distributed Image Recognition System that utilizes FaaS (Specialized PaaS) services and IoT devices to perform real-time face recognition on real videos recorded by the devices. Specifically, we developed this application using Amazon Lambda based FaaS and Raspberry Pi as an IoT device. Lambda is the first and the most widely used Function as a Service PaaS. Raspberry Pi is the most widely used IoT development platform. Main objective of this project was to gain experience in developing real-world, IoT data-driven cloud applications.

2. Design and Implementation

2.1. Architecture



Raspberry Pi (Python): It captures a continuous stream of videos which is used for real-time face recognition. For the face-recognition part, it sends a video frame in base64 encoded form every 0.5 seconds to the lambda, let's call it **Lambda 1**. We used lambda's new function URL feature to invoke it via the REST interface. In response, lambda returns the academic information of the recognized student in the frames. Pi gathers the outputs from lambda 1 and displays them in the terminal. We used the piCamera module in python to capture videos and FFMPEG to extract frames.

AWS Lambda 1 (Web Tier): This lambda function takes in the image in base64 form and invokes another lambda, let's call it **Lambda 2**. It gets the face recognition result from lambda 2 and invokes **Lambda 3.** Then, it gets the academic information from lambda 3 and returns that information back to the Pi.

AWS Lambda 2 (evaluation): This lambda function utilizes the actual machine learning code to perform the face-recognition task. When this lambda is triggered by Lambda 1, it receives an image from it and saves it locally. Then, it runs the trained Machine Learning model given to us to recognize the person in the frame. Once it executes it successfully, it returns the result back to lambda 1. This function is created using a docker image. The container image allows for the necessary libraries that the model uses, like pytorch and pytorchvision, to be contained inside the image without the need for installation in the lambda functions. We created the docker container image and uploaded it to AWS ECR. AWS ECR is a container registry that can be used to store the container images, using the container image uploaded to AWS ECR, the lambda function is created.

AWS Lambda 3 (information retrieval from dynamoDB): This lambda is triggered by Lambda 1. It receives a student's identifier from Lambda 1 and searches that student's records in the DynamoDB. Once it finds that student's information, it returns that information back to Lambda 1. We used the boto3 library to interact with DynamoDB.

DynamoDB: It is a key-value store containing information about the students. The table we created is indexed on the ASU ID of the students.

AWS S3: It stores the video captured by Raspberry Pi.

2.2 Concurrency and Latency

We used multithreading in Raspberry Pi to record the videos and extract frames concurrently. When the program starts, it continuously records videos of 0.5 seconds. Then the program creates new threads for frame extraction from each recorded video. These threads extract the frames, trigger the main lambda to get the academic information of the recognized person and print that information. Since the frame extraction threads work independently from the video recording, no frames are dropped.

On average, per frame, **end-to-end latency** from extracting the frame to getting the student's information back from lambda, pi takes **1.5 seconds**. Initially, our lambda used to take 20-25 seconds due to the cold start and 3-4 seconds afterward for the execution. To speed it up, we increased its memory from 512 MB to 3 GB. This small change provided us with a big improvement in terms of latency.

Since the time between two lambda invocations (0.5 seconds) is less than the time between getting their responses (1.5 seconds), AWS starts scaling out of lambda which again promotes concurrency. As per our experiments, we observed **15-25 peak concurrent invocations** of lambda.

3. Testing and evaluation:

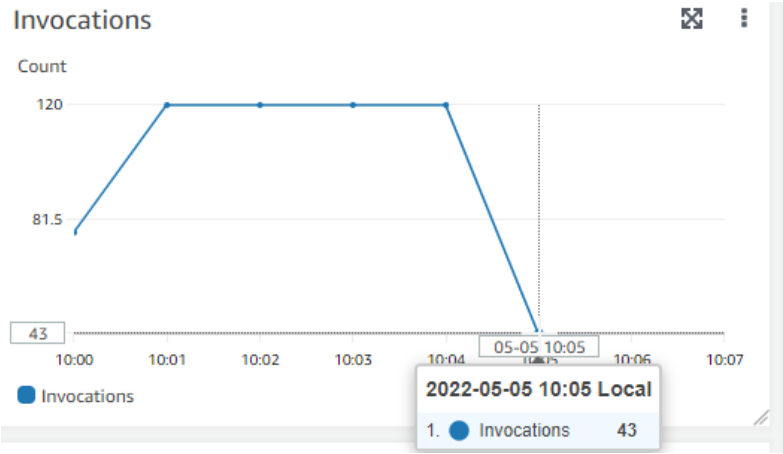
The resources from the us-east-1(Virginia) region are used. We used the free tier provided by AWS for all different services like lambda, ECR, S3, and DynamoDB. During testing, a video was recorded from the camera of the raspberry pi and each of our teammates appeared before the camera for more than 30 seconds. We checked the correctness of output, latency, concurrency of lambda functions, number of invocations of the lambda function, and the proper storage of video on S3.

- Proper storage of video on S3

We checked if videos of the entire 5 min duration (i.e. 600 videos of 0.5 s each) were stored on S3 or not. We made continuous movements before the camera for 5 minutes to analyze if there were any dropped frames. As per our testing, no frames were dropped.

- Number of invocations of lambda

To check if the raspberry pi was extracting frames from the video stream every 0.5 secs we checked for the number of invocations on the lambda console. For 5 minute of videos in which a frame is sent to lambda every 0.5 secs, there must be 600 invocations of lambda. This aspect was thoroughly checked to avoid incorrect lambda invocation.



The above screenshot is from the lambda dashboard. As we can see, from 10:00 to 10:05, we got 77, 120, 120, 120, 120, 43 invocations in each minute, summing up to 600 which was expected.

- Accuracy of face recognition

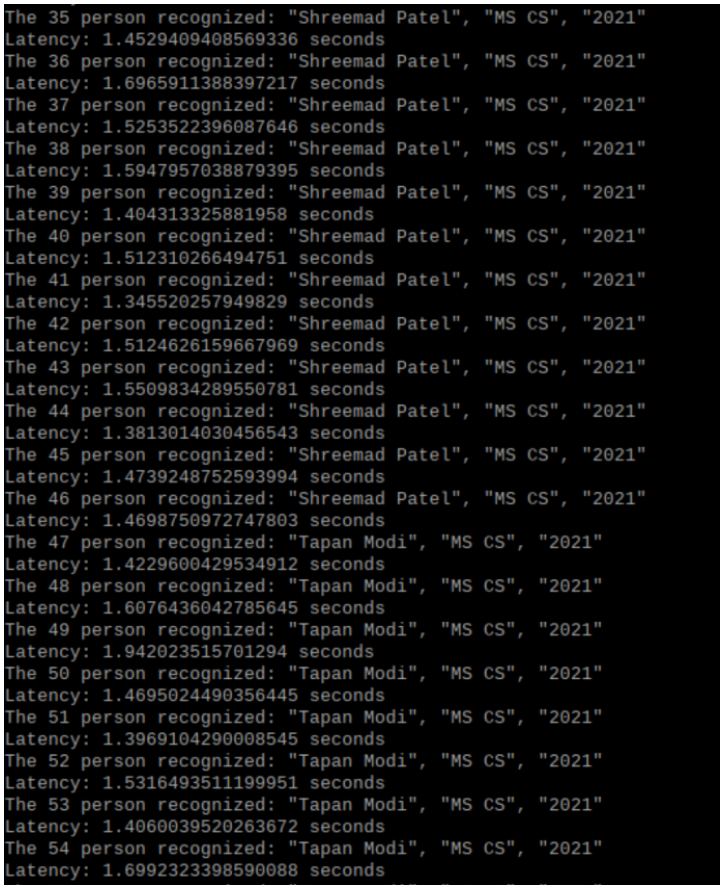
To ensure that the output of face recognition was correct more than 60 percent of the time, we first trained the model on our faces. We prepared 24 images of each of us for training and 3 images for validation. For the real-time testing, we divided the 5 minute duration into segments and presented our faces in order. We recorded the accuracy of output based on the segment of time. For example, we wrote code for checking the accuracy of the output returned to lambda between 1-2 min, 2-3 min, and 3-5 min. The face remained the same during the time duration. We observed that 80% of the results were correct i.e. out of 10 predictions, 8 predictions were correct.

- Autoscaling of lambda functions

To check if the lambda functions were auto-scaled correctly we checked the number of concurrent invocations in the lambda console. Since the auto-scaling is handled by AWS and we had no control over it, we verified that the max number of concurrently running lambda functions was greater than 3. Across all 3 lambdas, our implementation achieves 15-25 peak concurrency most of the time.

- Latency

We print the latency of the response with every output. Latency is calculated as the time taken by the application from sending the frame for evaluation to lambda and receiving a result from lambda. As we can see in the screenshot below, the latency stays in the range 1.3 - 1.6 seconds.



4. Code

1. Web tier lambda function

   The main function in this lambda function from where the execution starts is the lambda_handler function that takes an event as an argument. The arguments passed in the body while calling the function url are received by the lambda function in the event argument. This function acts as a controller between the evaluation lambda function and the dynamo db lambda function. The received base 64 encoding of the image is passed to the evaluation lambda function. It receives the evaluation response and passes the received evaluation result to the dynamo db lambda function. Finally, the received response from the dynamo db is sent back to the raspberry pi as json.

2. Evaluation lambda function

   This lambda function is responsible for face recognition of the received image. The lambda function is created from a docker image that is uploaded to ECR. The trained machine learning model is uploaded in the docker image and is used for recognition. The function decodes the base64 encoding of the image and stores the decoded image in the /tmp folder for the machine

learning model to access it. The evaluation is sent back to the web tier lambda function in the form of json.

3. dynamoDb lambda function
   This lambda function is designed and implemented to retrieve data from dynamo db based on the id and return it to the web tier lambda function. The function is implemented using the api provided by boto3.
4. Raspberry Pi - main.py
   - First, it initializes the PiCamera module and sets up its resolution to 1024 x 768 pixels.
   - Then, It starts recording videos of 0.5 seconds and performs frame extraction in separate threads. As it extracts frames, the image is sent to the web-tier lambda to perform the face-recognition task and get the academic information.
   - Once the videos are recorded and frames have been extracted, it will convert the H264 video into MP4 and upload them to S3 (upload_to_s3() function).
   - This file uses utils.py for some utility functions to do a POST call to lambda, call the S3 API to upload the video etc.
5. Raspberry Pi - utility.py
   - This file has a few utility functions which is used by main.py
   - post_call() - triggers lambda with image as payload. Then, it waits for its response and once it gets the response, prints it to the console.
   - upload_files_to_S3() - uploads file from given path to AWS S3

4. Running the code

1. There are 4 folders in the “code” folder - 1) dynamodb-lambda 2) evaluation-lambda 3) web-tier-lambda 4) raspberry-pi.
2. We have used the SAM CLI provided by AWS to build and deploy the evaluation lambda function. Its codebase includes a samconfig.toml that has all the deployment configs. The docker image for the function can be built using the "sam build" command. Before deploying the face-recognition lambda function, we need to train the model first to get a checkpoint file which will be used while testing. To train the model, we can just run the "python3 train_face_recognition.py" command. To deploy the docker image to AWS ECR we use the command "sam deploy –guided" which helps deploy the image step by step.
3. Since the web tier lambda function and the dynamo db lambda functions are relatively very small and consist of only one file, they can easily be developed, maintained, and deployed directly on the code CLI provided by lambda.
4. To run the code in Raspberry Pi, first, make sure that python 3.7 is installed and the terminal is currently in the raspberry-pi folder. Then we need to execute the following commands to successfully run the code.

```
python3 -m venv project-venv
source project-venv/bin/activate
pip install requests boto3 picamera
python main.py
```

Individual contributions - Onkar Pandit (ASU ID : 1222405119)

Design:

- I was primarily responsible for the design of the lambda functions that act as the web tier and the lambda function that performs face recognition. We designed the lambda functions to separate the important tasks of the application, keeping in mind auto-scaling requirements.
- We created a docker image for the lambda function that performs face recognition to accommodate the heavy libraries like PyTorch and PyTorch vision. I designed the code structure of the lambda function and the docker file.
- I contributed to the design of the docker container to be uploaded to AWS ECR. This docker container was pivotal to the application because it contained the machine learning model and its concerning libraries. The size of the libraries and the model do not allow it to be used in AWS lambda directly, hence creating a docker image and uploading it to ECR was the application design path we implemented. After doing some research on the best way to build and deploy the docker image I used the SAM CLI provided by AWS to build and deploy the container image.
- I designed the entire evaluation lambda function. The main challenge here was to integrate the given machine learning model code with a starter function in the lambda and pack everything inside a docker image.

Implementation:

- We used the boto3 library in python to use the API provided by AWS. After understanding the documentation of boto3, I implemented the internal calls between the web tier lambda, the evaluation lambda, and the lambda function that retrieves information from dynamo db.
- I implemented the web tier code where the lambda function took input from the raspberry pi in the event, decoded the base64 encoded image, and forwarded it to the eval lambda function. The order of execution was very critical here as the web tier cannot proceed to send arguments to the dynamo-db lambda function before the evaluation of the received frame is complete.
- I implemented the eval lambda function to integrate the lambda function handler with the provided machine learning model code. This function extracted the received base64 encoding of the image received from the web tier lambda and saved it in the /tmp folder for the machine learning model to access.
- I implemented the docker container code for the evaluation lambda function. This was an important step because I had to ensure that the size of the docker container did not blow up because of unnecessary libraries and files included in the container.

Testing:

- I tested the latency of the lambda functions. The end-to-end latency of the lambda functions was a critical part of the project. Considering that we are using a FaaS service provided by AWS, we had to modify the memory allotment to the functions to reduce the average latency including the cold start.
- I also tested the autoscaling and invocations of lambda functions. This step was crucial because if lambda functions are not scaled properly, it would make predictions delayed and if any lambda invocation was dropped, we would miss its prediction. So, I verified that we get a total of 600 successful invocations, and the concurrency of functions is at least 3.
- I also tested the docker container image to confirm that the model is being deployed correctly. This was an important step of testing because if the eval function lambda was not able to find the checkpoint file it would produce random outputs.

Individual contributions - Tapan Rajnikant Modi (ASU ID : 1222325026)

Design:

- I mainly designed the workflow from Raspberry Pi to AWS Resources. For example, we had to make a decision on how many videos to store in S3 and also how to extract frames from the continuous video recording. So, I did thorough research and read the documentation of the PiCamera module. With my findings and collaborative effort, we decided to store 600 videos of 0.5 second each in S3 and used FFMPEG tool to extract frames from the video.
- One other challenge in this project was to extract frames and record video simultaneously, meaning video shouldn't drop frames while frames are being processed. This definitely required parallel programming. So, I did some PoCs on multithreading and multi-process modules of python. From my experiments, we observed that multithreading achieved better performance so we decided to go with that approach.
- Additionally, I also contributed to resolving one issue of our architecture. Our default lambda setup was taking 3-4 seconds to return the response which is double of what is expected. I went through Lambda's documentation and searched on the other internet resources to figure out the cause and at the end I found out that the default lambda set up provides 512 MB of memory. So, I suggested increasing it to whatever memory we get in the free tier, which is 3 GB. As soon as we did that, our latency dropped below a second. Not just that, the cold start time was also reduced significantly.

Implementation:

- I worked mainly on the primary logic in Raspberry Pi. I used the PiCamera module to interact with Pi's camera. I implemented multithreading to perform frame extraction and video recording simultaneously.
- I also coded internal implementations of frame extraction and video recording. For frame extraction, I used the FFMPEG command. We decided to upload raw .h264 video in MP4 format to S3 because raw format is not supported on all computers. So, I implemented this conversion logic using the MP4Box tool.
- Moreover, I implemented the lambda to retrieve the academic information of the predicted person from DynamoDB. I coded this function in python. So, I used the boto3 library to connect to the DynamoDB. Since we are querying the primary index of the tables with the ASU id of the student, I used KeyConditionExpression of boto3 dynamoDB client for the search.

Testing:

- I tested the post_call utility function in Raspberry Pi. We had to make sure that whatever response comes back from lambda, we should be able to handle it. For example, if the lambda execution was successful, the utility function should be able to extract the information in the format we want. On the other hand, if the lambda execution was not successful, the utility must show an error message to the user. I tested all these edge cases diligently by simulating them.
- I tested the concurrency in the Raspberry Pi. I made sure that extracting frames and recording of videos don't affect each other. Since we are extracting frames every 0.5 seconds, we also don't want these frames to be extracted late. So, all the post calls to trigger lambda also happen in separate threads. This multithreading approach made it possible.
- For the dynamoDB lambda function also, I tested all edge scenarios such as what happens if credentials are wrong, what if the table is missing, what if the student is not added to the table etc.

Individual contributions - Shreemad Sanskarbhai Patel (ASU ID : 1222713687)

Design:

- I also contributed to setting up the triggers for the lambda functions on AWS. Here, we had to perform several experiments to identify the best trigger from API Gateway, S3, and Function URL in lambda. The decision was made on the basis of ease of implementation, deployment, and latency.
- I also contributed to the design of how we would trigger lambda from Pi. We tried a few approaches. I mainly performed PoCs on newly introduced lambda function URLs which allow us to trigger lambdas via the REST interface. Since it was a new feature, I also validated its stability by doing load testing. Finally, we decided to proceed with this approach as it made our implementation a lot easier.
- I contributed to the setup of training the model. The requirement was to get an accuracy of more than 60 percent. On the initial run, we were only able to get to 55 percent accuracy. To get good accuracy on the model, instead of training on photos with the same pose, I suggested training on photos with different angles. So we rotated our heads in several directions before taking photos. These photos provided the model with richer information about each of us's faces.

Implementation:

- I contributed to the implementation of some of the utility functions in raspberry pi like the post_call function which calls the function URL of the web tier lambda with the extracted frames and the upload_files_to_s3 which sends the video files to S3. I used the boto3 library to interact with both S3 and lambda.
- I contributed to the implementation of the eval lambda function. This function extracted the received base64 encoding of the image received from the web tier lambda and saved it in the /tmp folder for the machine learning model to access. I mainly implemented the configuration files and docker commands for a proper build.
- I systematically wrote all the logs in the different lambda functions. These logs were designed to indicate the progress of the lambda functions and the locations of errors if any occurred. We then monitored our lambda function processes in AWS Cloudwatch to monitor the sequence of execution.

Testing:

- I checked the recorded videos to ensure that no frames were dropped and faces were shown in proper brightness and contrast. The color profile of the camera was necessary because by default the camera's images were a bit darker which was affecting our model's predictions. So, I tested multiple times to make a few small changes in its color profile to show the person's face clearly. Additionally, the model also required images in 160x160 pixels.
- I also tested the accuracy of the training model. So, we divided the 5 min video into 3 segments. In each segment, one of us would be in front of the camera. So, the prediction should be for the same person for that segment. I gathered the results for each segment and calculated the accuracy of the model.
- Additionally, the upload_files_to_s3 utility must be able to upload the file to s3 and handle error scenarios. I tested it on various cases such as successful upload, missing files, and wrong credentials and validated its behavior to the user. I generated these situations forcefully for the testing.