CSE 546 — Project2 Report

Arindam Jain	*Panemangalore Srikanth Kini*	*Bhavani Mahalakshmi Gowri Sankar*
ajain243@asu.edu	*pkini2@asu.edu*	*bgowrisa@asu.edu*
1223458106	1222310674	1222186719

1. Problem statement

In this project, we are creating a distributed application making use of PaaS services and IoT devices to conduct real-time facial recognition on videos captured by the devices. This application will be built utilizing an AWS Lambda-based PaaS and a Raspberry Pi-based IoT. The earliest and most extensively used Function-as-a-Service PaaS is Lambda. The Raspberry Pi is by far the most popular IoT development platform.

To solve this problem, we use Lambda (Function as a Service PaaS) , Simple Storage Service(S3), DynamoDB, Raspberry Pi-based IoT, and the AWS SDK(boto3) that provides access to these resources and the capability to manipulate them, moreover, it is an important problem as we need to meet the real-time demands of the application, handle multiple requests concurrently and return the results as rapidly as possible and without losing any requests. We use it to provide real-time image recognition to the consumers of our application.

2. Design and implementation

2.1 Architecture

Amazon Web Services (AWS) services used in the project:

1. AWS Lambda

AWS Lambda allows users to run code without the need to provision or manage servers. Users only pay for the computing time that is used; when the code is not executing, there is no price. Users can execute code for nearly any form of application or backend service using Lambda, and do not have to worry about management. Simply upload the code, and Lambda will handle everything necessary to execute and grow it with high availability. Users may set up the code to be triggered automatically by other AWS services, or can call it directly from any web or mobile app.

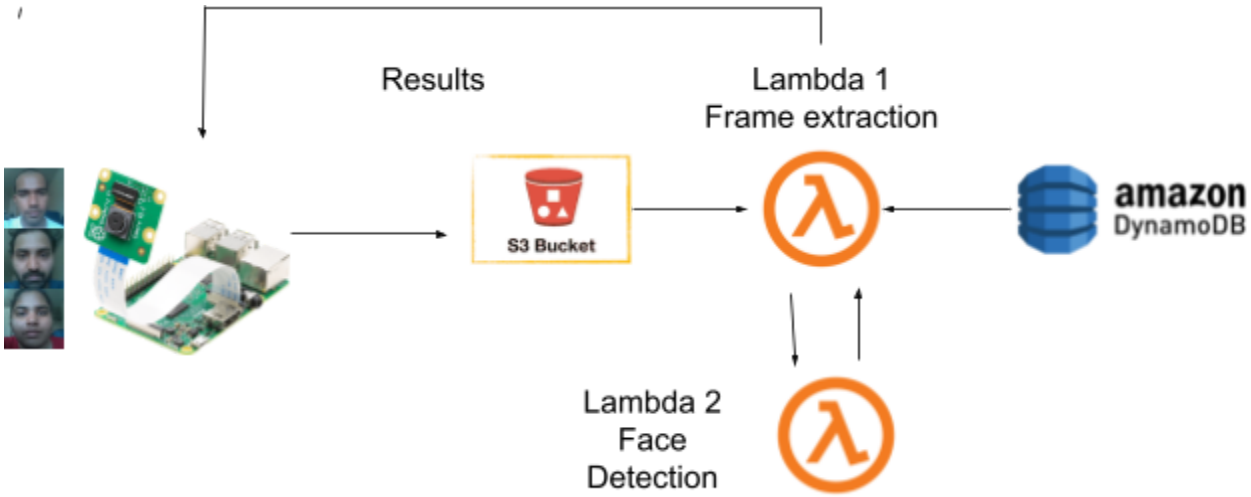
2. DynamoDB

DynamoDB is a nonrelational database service that scales to any size. Customers can use DynamoDB to offload the administrative burdens of running and scaling distributed databases to Amazon Web Services (AWS), so they don't have to worry about hardware provisioning, setup and configuration, throughput capacity planning, replication, software patching, or cluster scaling.

3. AWS Simple Storage Service (S3)

For consistency, we use S3 to store videos recorded from the Pi. Through application programming interfaces, S3 enables scalable storage. Any type of files may be stored in S3 buckets using the (key, value) object format.

Architecture Diagram



The Raspberry Pi records the videos using the attached camera module. The videos are uploaded to the cloud in S3 buckets in AWS. The AWS Lambda functions contain code to extract image frames from the video, use a deep learning model to classify the image, retrieve the relevant details from DynamoDB and return the results back to the PI. The cloud performs face recognition on the collected videos, looks up the recognized students in the database, and returns the relevant academic information of each recognized student back to the user.

2.2 Concurrency and Latency

The script used to record videos makes use of the `raspivid` command to capture videos in 160 x 160 resolution in a finite loop. The loop runs as many times as the required time for capturing 2 videos every second until the time given as command line parameter to the script. Two lambda functions, one to extract the frames, another lambda function to classify and retrieve the details from DynamoDB are used in the architecture.

The lambda functions use an invocation type of “RequestResponse”, which ensures that they function synchronously similar to REST APIs, but since these invocations are made via threads, the lambda functions can have parallel executions and also work concurrently. As the loop runs continuously recording videos, and creating threads that invoke the lambda functions synchronously, the end-to-end latency of each request is minimized.

3. Testing and evaluation

We tested several different implementations of the lambda functions before settling on the most efficient one. Our final implementation involves two lambda functions, one that performs the frame extraction out of the videos uploaded to S3 and another one that uses a deep learning model to perform prediction and retrieval of results from DynamoDB. Our first implementation made use of SQS for queueing the results from lambda and sending it back to the Pi in First In First Out(FIFO) order. This took

about 8 seconds for each image and was highly inefficient. We replaced SQS with direct invocations between the Pi and the lambda functions. This seemed to improve the time by a couple of seconds and the latency was around 5 seconds. In an attempt to improve it further, we combined the logic of model prediction and getting results from dynamo in a single lambda function, which then returns back the predicted student details to the user. This was our most efficient implementation that has a latency of around 1 to 1.5 seconds. We evaluated the model by testing for different amounts of time intervals and recording the latency for each request. Additionally we checked the time taken by each lambda function to execute the code and return the final result.

Another major improvement we did was to increase the accuracy of the model from 88% to 97% by changing the optimizer to Adam, batch size to 12 and choosing the least loss error in validation data as the best model

4. Code

push.py	It contains the code for recording videos and sending 0.5 second videos to the S3 bucket and also get back the results from the DynamoDb to the Pi.
lambda_1.py	It contains the code used in the first lambda function that includes downloading video from S3, extracting frames and invoking the second lambda.
Image recognition training & validation	This folder contains the deep learning AMI code used for training and evaluating the model.
lambda 2 docker image	This folder contains the dockerfile and handler file containing custom code to classify image frames using the deep learning model.

Table 1.

To run the code on the PI,

Install pip:- **sudo apt-get install python3-pip**

Install boto3: **pip3 install boto3**

To run the push script in Raspberry Pi , **python3 push.py <time_in_minutes>**

The lambda_1.py file contains the code for the first lambda function which can be deployed as is.

The lambda 2 docker image contains the dockerfile to set up the image for the second lambda function as below:

docker build -t docker-image-for-lambda .

docker push <user_id><region>.amazonaws.com/docker-image-for-lambda:lates

5. Individual contributions

Arindam Jain (ASU ID: 1223458106)

I. Design:

My responsibility in this project involved setting up the entire Raspberry Pi from installing the OS, to setting up the wifi module, to setting up the camera module and then implementing the logic to record videos in push.py script that is run on the Raspberry Pi and save it to S3 bucket. I wrote the logic code to capture videos from the Raspberry Pi for a given time interval and upload the videos to S3 every 0.5 seconds. I also worked on improving the accuracy of the model that is located in the second lambda function of the application which helps in face detection.

II. Implementation:

My main responsibility in this group project was setting up the Raspberry Pi to connect to the cloud services and implementing the initial script push.py which is used to capture videos and get results from lambda.

I worked on booting the Raspberry Pi, setting up the wifi, camera module to capture videos using raspivid library to save the video in h264 format thereby ensuring end-to-end continuous video recording and saving, and enabling SSH. The push.py script records videos indefinitely on a loop based on the given time of execution and uploads videos of length 0.5 sec to the S3 bucket. The push.py scripts also output the person's name and academic information in the specified format.

Another major part I worked on was improving the accuracy of the model with better pictures and taking the least validation loss as the key metric to get the best model. The model was trained using Adam as the optimizer and changed the batch size to 12 which improved the validation accuracy from 86% to 97%.

III. Testing:

Our initial idea was to extract frames from the Raspberry Pi and route it to the lambda function and upload videos in S3, however it turned out that computation power on the Raspberry Pi is not very fast and it was increasing latency and our team decided to extract frames on the lambda function and upload 0.5-second videos to S3 and extract frames from the middle of the video i.e. 0.25 sec on the lambda function, which improved the latency.

I worked on improving the accuracy of the machine learning model. While training I observed that even though we were able to achieve 100% accuracy, the model was not performing well in real-time so I tested with different parameters i.e. Optimizers (Adam seems to be the best) and batch size of 8,12,16, and 32 (12 seems to be the best) which helped our team achieving 97% validation accuracy and 100% training accuracy and selected the least validation loss parameter as the best model for testing.

Panemangalore Srikanth Kini (ASU ID: 1222310674)

I. Design:

I worked on the integration between the different lambda functions as well as creating the custom docker container used to build the pre-trained model. I optimized the end-to-end latency of the architecture by reducing the process time between the different components. I worked on designing the concurrent execution of the lambdas as well as returning the results back to the PI after classification using threading and concurrency.

II. Implementation:

As part of building the custom docker container, I extracted the required code from the given AMI to classify image frames received from the first lambda. I implemented the handler script containing the face_recognition_handler, for the second lambda, by writing necessary functions that made use of the deep learning library to classify the image. I bundled up the required dependencies and created the dockerfile to containerize the custom model for the purposes of classification. I took the images captured from the PI and used them to train and validate the custom model by using the training and validation scripts provided.

Then I worked on integrating the different lambdas, initially using S3 as a trigger for the first lambda that performs frame-extraction. However, this resulted in high latency, after which I called the functions from within themselves using a request-response invocation. I also set up multi-threading in the initial push script from the PI, that is used for recording videos, to handle the concurrent executions of the lambda functions.

I modified the lambda functions to work in a synchronous manner by returning the result of the 2nd lambda back to the first lambda, and finally returning the desired result from the first lambda back to the PI console through the threads.

III. Testing:

I tested the second lambda function by deploying the docker container and invoking the lambda with test cases through a script. I used the logs from Cloudwatch to fix any issues that were encountered. I tested the first lambda function by invoking it through a script and passing a saved video in h264 format, directly as a parameter. Then I added synchronous invocations using the “RequestResponse” type, from within the lambdas. The first lambda performed the frame extraction and invoked the second lambda passing the frame directly as a base64 encoded string. The second lambda classified the frame using the model, retrieved the desired results from DynamoDB and returned the results back to the first lambda. The first lambda then returned the results back to the PI. I tested the end-to-end time taken by calculating the time for each lambda execution as well as overall time taken.

Additionally I tested the concurrency of the lambda functions by using the push.py script, which runs a loop to record videos. I added threading in the script to parallelize the upload of videos to S3, and the lambda invocation to extract the frames. I observed that the lambdas showed correct concurrent execution with multiple invocations in a 1 min time frame. In order to accommodate the additional overhead of writing the recorded videos of 0.5 seconds to the file, I added the logic to run a loop on the basis of the number of videos per minute, given the time as a command-line parameter.

Bhavani Mahalakshmi Gowri Sankar (ASU ID: 1222186719)

I. Design:

I worked on uploading videos from the raspberry pi to the s3 bucket, extracting the frames and sending the end result back to the user. I also worked on using SQS as a buffer between the pi and the lambda functions but we had to scrap the implementation as it was not an efficient way to send the results back to the Pi. Also, the latency was much higher compared to what was expected. So, we removed SQS from our architecture and set up direct invocation/communication between the Pi and the Lambda functions.

II. Implementation:

I began by writing a python script to extract frames on the Pi using ffmpeg. We later had to move the logic to the Lambda function as it was much faster and supported concurrency. Frame extraction needs ffmpeg to be available in the lambda environment. This had to be done by uploading an executable zip for the ffmpeg package to an S3 bucket. The ffmpeg executable was made available by adding it as an additional layer to the lambda function. I also set up triggers for the lambda function to pick up new objects that are being added to the S3 bucket. I also wrote a python script that uploads the recorded 0.5 second videos to S3. The current implementation works by recording and uploading videos to s3 which triggers the frame extraction function, extracting the frames and sending them to another lambda function that predicts the face image which then sends the results back to the user.

The other part was to store the student data in DynamoDB and write a function to read it based on the student name. I created a Dynamo table ‘student’ with ‘name’ as the partition key so that it can be used for querying purposes. Initially, I created a separate lambda function to implement this but in order to save latency, we moved the logic to the second lambda function that performed the prediction.

I also worked on capturing the images of our project mates. We tried training the model multiple times with different sets of images to achieve the best accuracy. Our latest model had 50 images each for training and 10 for validation using the `raspistill` command.

III. Testing:

I initially worked on implementing 3 different lambda functions, for frame extraction, using the model to predict the student name and getting the student details from DynamoDB. This implementation took approximately 5 seconds for each result. So, we had to move on into combining model prediction and DynamoDB retrieval into one lambda function. This brought down the latency significantly and we ended up with approximately 1.5 seconds per image.

Another implementation was using SQS to queue the output from lambda functions and make it available to the Pi. This meant constant polling from the Pi which would take up significant processing power of the Pi. This also increased the latency by one or two seconds. So, we ditched SQS and made the Pi directly receive the student details from the second lambda function.