CSE 546 — Project 2 Report

Beverly Winebrenner, Daniel Evans, Venkataramana Balaji Rajendran

1. Introduction

Given the many security and convenience use cases for facial recognition, a system which can rapidly identify faces and confirm identities has many real world applications. To begin with the real world applications, security may be one of the largest applications of real time facial recognition. From airports and border crossings to concert and school entries, security can be greatly enhanced where national security and public safety is involved. Persons of interest can be immediately flagged and addressed to avoid issues from occurring in the first place. In addition to these uses, convenience is another aspect of facial recognition that will be utilized in the future. From using facial recognition to access an atm to paying for public transit, facial recognition can offer users a convenient way to access tickets and their personal accounts without the need of other physical systems. The only way to allow for these uses is to make not only an efficient facial recognition system, but also one that is scalable and readily accessible.

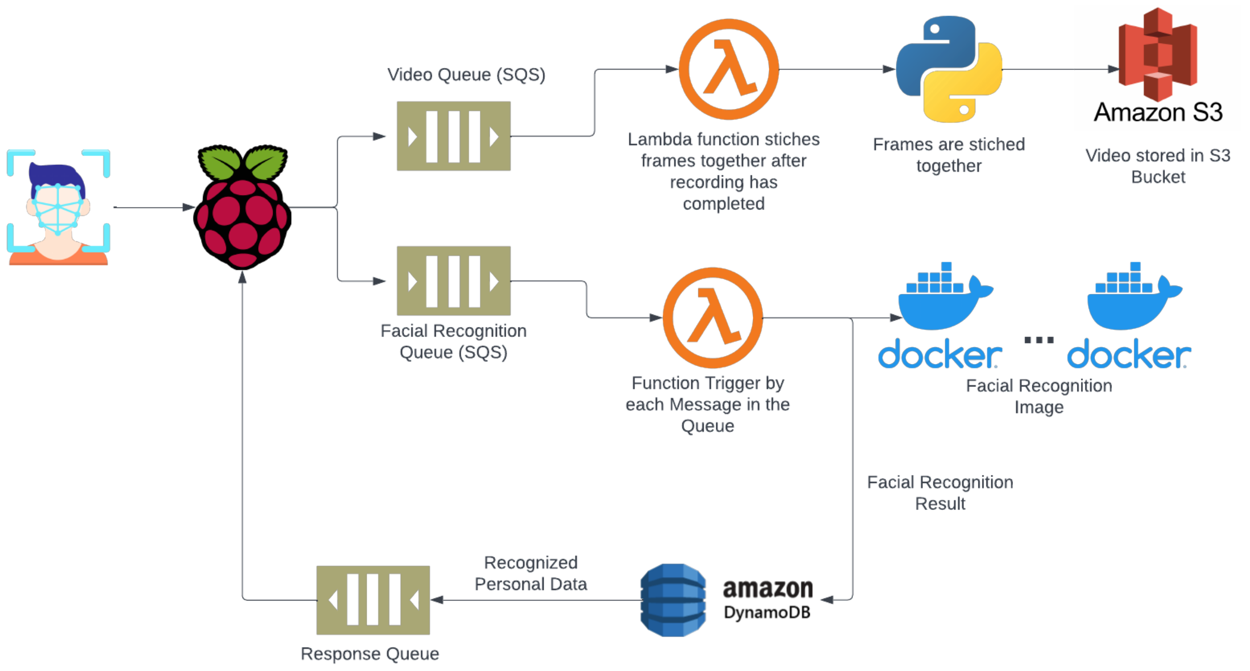
2. Problem Statement

For this project, we are tasked with implementing a distributed application that utilizes PaaS technologies and IoT platforms to perform real time facial recognition. The PaaS platform being used is AWS Lambda and the IoT platform used is Raspberry Pi. This system includes identification of faces every 0.5 seconds from the Pi and returns the identified person's information that has been stored in DynamoDB. In addition to this, the overall video recorded is stored in S3.

3. Design and Implementation

For this project, we were tasked with implementing facial recognition through hardware (Raspberry Pi) and software (AWS Lambda) so that it is efficient and scalable. The Raspberry Pi sends images to the cloud via AWS SQS which automatically triggers Lambda functions to take one message per Docker container which allows for a fast and scalable approach to processing and assigning identification. Once an assignment has been made, the information relating to the identification is received from a DynamoDB query and is returned to the Raspberry Pi for displaying. While the Pi is recording, frames are sent to another SQS queue every tenth of a second. Once the video is complete, an EC2 instance will grab the available images from the queue, sort them by time, and stitch them together to form the original video and save the video to S3.

a. Architecture


Video Queue (SQS)
Lambda function stiches frames together after recording has completed
Frames are stiched together
Amazon S3
Video stored in S3 Bucket
Facial Recognition Queue (SQS)
Function Trigger by each Message in the Queue
docker
docker
Facial Recognition Image
Facial Recognition Result
Recognized Personal Data
amazon DynamoDB
Response Queue

b. Autoscaling

Autoscaling is achieved in this system through the use of AWS SQS and Lambda. From the Pi device, facial recognition function is triggered via Lambda which is set to trigger from any existing messages within the Facial Recognition Queue. From there, each lambda function spins up a Docker container which will process the individual image and remove it from the queue. This allows for automatic scaling regardless of the frequency or number of images received at any given time.

4. Testing and Evaluation

When testing we are considering four main criteria: correct output, correct S3 bucket contents, correct Lambda autoscaling, and quick processing times.

To satisfy the correct output, every provided user image must return the correct face recognition result as output. Correct S3 bucket contents can only be satisfied once the video has completed recording and the entire video is present and plays as expected. The overall recognition result was considered to be achieved between 50 to 70% accuracy. Autoscaling of the Lambda functions is satisfied if the number of instances processing images scales to the largest quantity possible as defined in AWS's console. Quick processing time will be met if our app can process the maximum concurrent requests within the period of the ten minute video.

To confirm testing is successful, approximately 6000 frames should be transferred from the Video Queue to the EC2 instance to be stitched together to create a 10 minute video with 10 frames per second. To confirm testing for the Lambda functions is successful, all frames related to recognition should be sent to the Facial Recognition Queue (number of messages in the queue should be approximately 300 frames). The DynamoDB should be queried that number of times and each entry should be printed back to the user on the Pi for each recognition and data retrieval completed.

5. Code

1. pi_script.py
    a. This file is a Python script designed to be run on the Raspberry Pi locally. The script uses the picamera Python library to capture frames approximately 10 times every second. For every 0.5 seconds, a frame is sent to the Facial Recognition Queue to be processed by Lambda via SQS.
2. video_creator.py
    a. This file is a Python script that is deployed as an AWS Lambda function. This function is triggered by an http endpoint. Once triggered, the code pulls messages (frames) from the Video Queue and, once it has checked several times and each of those checks has received no messages from the queue, the script will stitch all of the available frames into a video of length approximating 5 minutes (during testing) or the length of recording done during face recognition. This function has an additional custom lambda layer (arn:aws:lambda:us-east-1:954573623204:layer:myOpenCVLayer:2) that includes the installation of opencv-python and other required python libraries.
3. app.py
    a. This is the application tier of this project. This is also a python script deployed as an AWS lambda function on top of a custom container image. The container image loads the updated model and other necessary components and libraries needed for the application environment. A new instance of this lambda function is triggered for every new message (image) in the prediction request queue. The lambda_handler() function reconstructs the image from the binary code message and uses the model to perform prediction. The model output (student name) is used as the key to retrieve student details from the DynamoDB table. The retrieved details are sent to the response queue.
4. Dockerfile
    a. This file contains the sequence of steps to be carried out at the time of building and running the lambda container. These steps involve copying files to the container image, installing the required packages and triggering the application lambda function (app.lambda_handler) at the end.

6. Installation and Running

To begin running the code for the Raspberry Pi operations, the repository needs to be cloned on the edge device and the command “python3 pi_script.py'' must be executed. This code will then execute three separate threads which will one, begin recording videos and sending those frames to the video queue and facial recognition queue, two, begin receiving results from the result queue (which returns from Lambda after retrieving information from DynamoDB) and writing the information locally, and three, print the results of the facial recognition in order of how the frames were originally sent.