CSE 546 — Project2
Report

Bishnupriya Pradhan(1222307502), Karan Dabas(1219496811), Swetaswini Nayak(1219505573)

1. Problem statement

In this second addition of our cloud project, we developed a real world face recognition system that utilizes two main components: the IoT device (Raspberry Pi) and cloud application that uses platform as a service components (AWS). The entire application was built using AWS FaaS services (a special subset of PaaS) among others. The FaaS stands for function as a service, Lambda function is the prime example of it. We make use of a variety of tools provided such as storage service (S3), faas service (Lambda), messaging services to transfer output across different platforms (SQS) and NoSQL table that stores academic information of recognized students (DynamoDB). The entire framework starts with a camera module attached to the IoT device that captures video of the person in front of it and sends that video to the cloud via raspberry pi. The cloud processes this video in two stages, first it extracts frames from the supplied video and pushes them to the second processing stage where they are fed into the on demand classifier function to be classified. At this time the function gets the classification result and queries it on the NoSQL table to get complete information of the person (including their major, type and name). The results from the table query are pushed back to the IoT device via messaging system and are consumed by it to display on screen at device end. The use of cloud components is required because of their ability to adapt to scaling out or in. It also makes the system on-demand and deals with large volumes of incoming video files without waiting for manual intervention. Fig-1 shows the overall system design, this approach with all its components gives us all the key characteristics of the cloud system and achieves the most reasonable implementation and gives the best possible mixture of cloud + IoT components with minimal latency and real world benchmark standards.

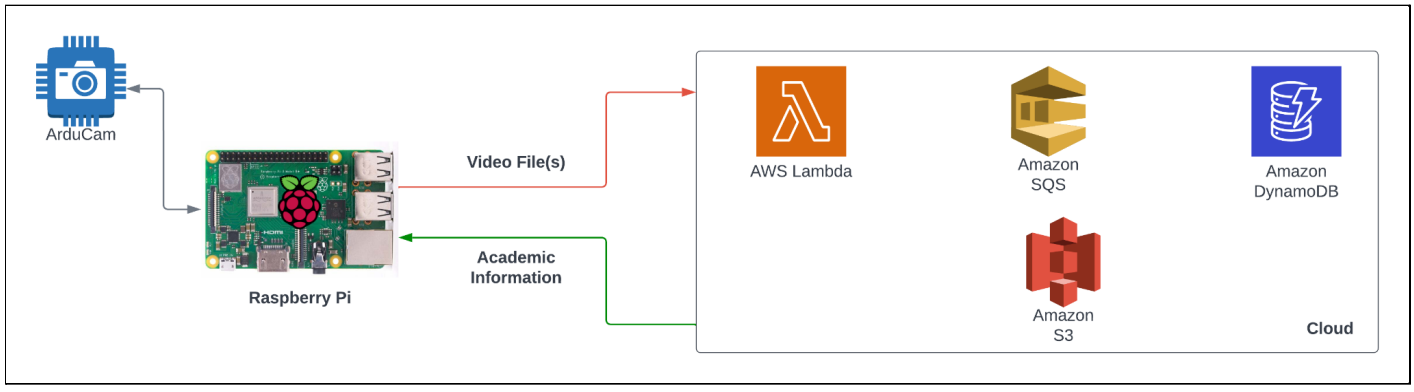

Fig-1: Overview of cloud application design.

2. Design and implementation

2.1 Architecture

Complete system architecture with all its connections and communications can be seen in Fig-2, on the IoT device we have two main functions: (i) Capture the videos in 0.5 seconds time frame and send it to the S3 storage on cloud for processing. (ii) Consume all the incoming results from the SQS that acts as a bridge between the cloud and IoT device and display them. At the cloud side, we have various working parts, the first one being the S3 storage that hosts all the incoming video files from IoT devices and model weights for the classifier, two lambda environment setup files in zip format and a frames folder that keeps that frames extracted from the uploaded videos. Once the video is uploaded to S3, the second procedure starts as we are making use of Lambda function that gets triggered by the new object creation and takes that event (video) and processes it using the CV2 python library to give us the frames from the video. These frames are written to a separate folder called frames in the same S3 bucket. Once we start receiving frames (.png) it triggers the second Lambda function that hosts a trained PyTorch classifier for face classification, as it feeds on the frames one by one it produces a classification for each frame and query that to the DynamoDb to get further details of the identified person. This querying is done within the classifier lambda function and as a final output this is sent to the SQS messaging service with FIFO setup to be consumed by the IoT device.

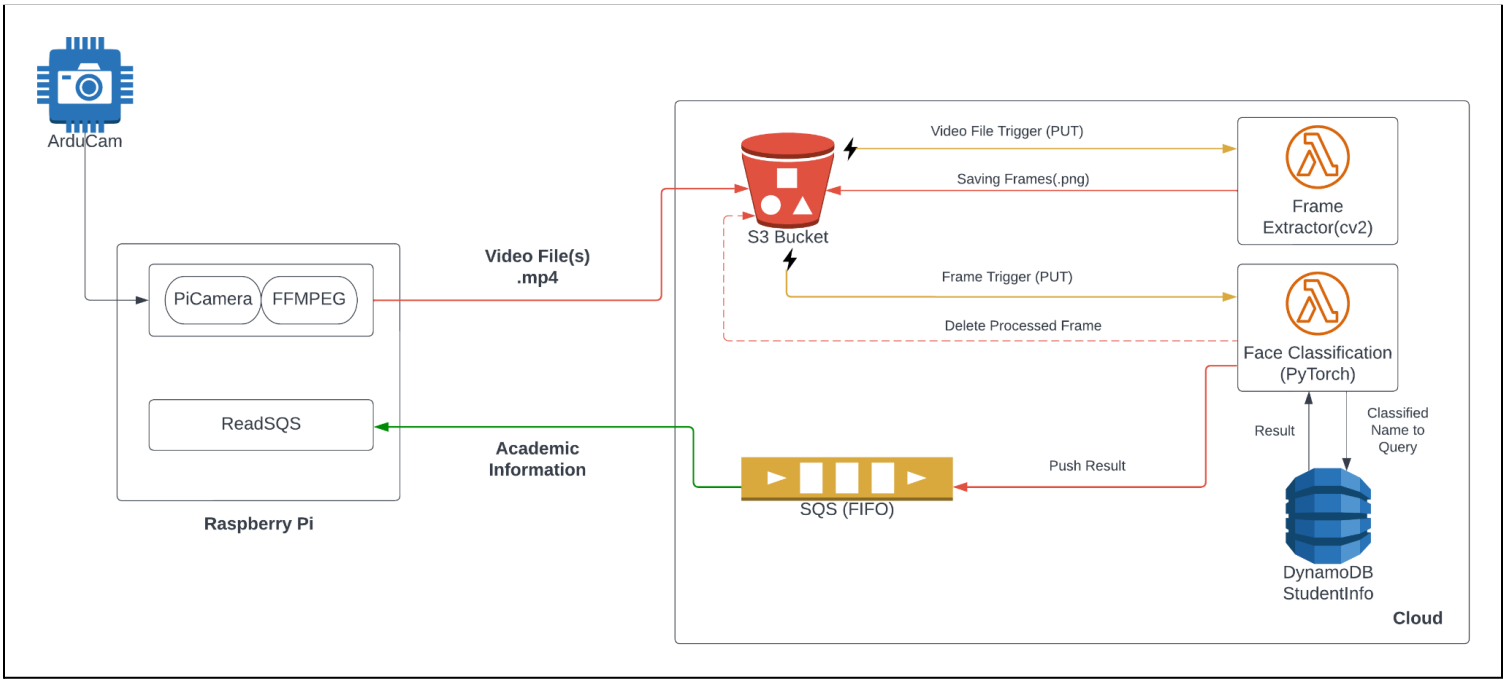

Fig-2: Architecture and workflow of the entire cloud application.

2.2 Concurrency and Latency

We achieved concurrency at two levels. At IoT device, we make use of two separate programs, one that records the videos in intervals and sends it to the cloud via S3 and another program that spawns 50 parallel threads to read/consume all the incoming messages from the SQS queue as fast as possible. The latency is calculated from the point in time when the video file reaches the S3 to the point of time the IoT device receives a result for the specific frame from the video file. We simply take the time difference between these two to get the overall latency. This latency is at frame level, which means for each result we get from the SQS (which corresponds to a certain frame from the video uploaded) it gets a corresponding time difference. To keep track of the time at video and frame level, we made use of the filename as the string datetime object of UTC time. The choice of UTC is done to avoid all the time difference problems between the IoT time zone and cloud infra time zone. The two lambda functions that we have for frame extraction and face classification do run in concurrent manner via cloud provider's own autoscaling as they increase the number of images of the function and do load balancing across them automatically.

2.3 Autoscaling

As oppose to project-1, where we made use of infrastructure as a service components like EC2 that required for further autoscaling algorithm from the developers side, this project does not. As we are making use of PaaS systems we only deal with the workings of the application and the rest is taken care by the cloud provider at the backend this includes on demand scaling and resource provisioning. This is only restricted to the cloud component of this project as for the IoT side, we are making use of additional techniques to get minimum latency through parallel threaded consumption strategy where we spawn 50 threads that all read incoming messages from the SQS.

3. Testing and evaluation

Starting with the provided PyTorch classifier that we trained on our own dataset. We started off by using the given camera module to generate data for training. We recorded two videos of 1 minute each for every group member with slight face movement and head orientation in front of the uniform white background. These videos were recorded at different lighting conditions with normal room lighting and other with special flash + room lighting to give more details of the face. We extracted frames from these videos and used it to train our model but soon realized that the model was not performing well and was only giving borderline accuracy of around 70%. We then decided to make use of a special opencv function called cv.detect_face() to extract just the face from the entire image and re-trained our model now with a total of 120 images per person in training and 20 images per person for testing. This makes our total train set to be 360 images and test set of 60 images. This new face extracted images we got a **training accuracy of 97%** with a **testing accuracy of 98%**. To note that we only did this during the model training stage and for the real time application we are still using the complete images. After this we proceeded with our lambda functions, as we were testing our lambda functions we noticed that whenever we start the function for a certain event for the first time it takes some time as opposed to the case after it. This is where we were introduced to the lambda cold start problem, using our heaviest lambda function that uses a Pytorch library to do classification we show the following difference between a cold start versus the warm start.

Cold Start (time to process an event)	**Warm Start** (time to process an event)
19.9 sec	0.698 sec

Apart from this we also calculated the overall accuracy at the real time application, the percentage of number of correct people over all the results in a certain video. We got a 98% result on the videos with the only exception being that our application gets random results in between the transition of person one and two and we understand this could be because of the motion blur or partial faces during person transition.

As for the lambda function metrics over 5 min run, on pytorch classifier we are getting an average duration of 680ms, number of invocations close to 600, success rate of 100% and concurrent executions around 258. For our opencv frame extractor we are getting an average duration of 1041ms, number of invocations of 100 and success rate of 100% and concurrent executions of 5.

4. Code

Model training: We utilized the given AMI to generate an EC2 machine to train our model with provided codebase, the only addition to this was our special face extractor script that helped us train the model in more acceptable accuracy of 97%. This is a data processing script that gives frames of just a person's face from the recorded videos for training. The script is called **extract.py** it takes two arguments one is the input video file location and the other is the destination folder where the extracted frames will be saved. It contains two functions a **main()** that reads the input video file and extract the normal frames from it and then applies the **cv.detect_face()** function on the image array to get the extract face location in the frame that I use to crop the image and lastly resize it to required 160x160 dimensions for model training. The other function is **getFiles()** that takes one argument of the location and os file path for it.

Lambda-1 (frame extractor): it only contains one .py file that has the set lambda function in it. The python file imports cv2, datetime and os libraries. It has the main **lambda_handler(event, context**) function that gets executed whenever an event occurs based on the set triggers. With this function we make calls to helper functions such as **download_video(bucket='', key='')**: that takes bucket and key as input and downloads the video to the local /tmp directory of the lambda, **save_s3img(img,filename)**: it takes two arguments image/frame array and filename string and stores the frame to the desired folder with this filename in the set s3 bucket, **extract_frames(video_path,base_name,dt_string)**: this the function that reads the downloaded video and extracts frames from it and calls the **save_s3img()** function for each frame.

Lambda-2 (pytorch classifier): it has multiple python files, the main being the lambda function file called main.py that contains **download_image(bucket='', key='')**, **download_model(bucket='', key='')**, **download_model_labels(bucket='', key='')** all these files download the image, model weights and model labels to the local directory (/tmp) of the lambda container. The lambda function is called **lambda_handler(event, context)**: that takes event and context as input and extracts the frame(.png) name and bucket from the event json and invokes the download_image() function along with

download_model() and **download_model_labels()** function if not already downloaded to the /tmp folder. Apart from this we also have **classify_image(model_path,labels_path, img,bucket,key)**: that creates builds the model using the provided build script and other model python files and loads the weights from the downloaded weights file and runs the provided image/frame to the classifier and produce label. This label is then sent as part of a query to DynamoDb to fetch complete academic information which is returned as a json and we dump this into string and call in the **send_result(result,filename)**: that sends the result along with the filename to the SQS.

Raspberry pi (setup): At IoT side, we have two python scripts: (1) startCameraCapture.py: that is responsible for capturing the videos from pi camera module and converting it to the .mp4 using ffmpeg and upload the new video to S3 bucket. All of this is placed inside an infinite loop to keep on recording and transmitting data to the cloud. Once a video is uploaded we also delete the local copy of the video to save space on pi device. (2) readSQS.py: it acts a consumer script that has only one task, to read all the incoming messages in the SQS and calculate the latency from the time the file was uploaded to s3 information of which is under message author tag and current time at which this message was consumed by the script. This script executed 50 parallel threads that all run the same operations of consuming SQS and latency calculations individually and print it to the terminal screen.

Project Setup: Before we begin, make sure you generate a new EC2 from the provided AMI(ami-016931ea066955c9d) for training the model. We will be using the same EC2 instance to create our required lambda python environments.

Lambda Function-2 (Pytorch Classifier)-- Launch the instance from above AMI. After this access the instance using ssh. Make sure that you download the accessKey file.

SSH command: $ ssh -i <accessKey file> ubuntu@<public id of instance>

After this create a new folder in Desktop and cd to it and follow the below commands,

$ python3 -m venv venv

$ source venv/bin/activate

$ mkdir code

// place all the scripts including build_model.py and main.py(lambda script) and model build folder in this code directory and unzip_requirements.py file provided in submission.

// deploy.sh in the same directory as code folder

$ pip install torch==1.4.0+cpu torchvision==0.5.0+cpu -f https://download.pytorch.org/whl/torch_stable.html

//after this run the deploy.sh file, but make sure that before this you have aws cli setup on the instance. See reference: https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html and add your account to it. Also make sure that you have a simple lambda function setup on AWS and an S3 bucket. Edit their names in deploy.sh and main.py file that uses lambda,S3 and S3 respectively.

$ sh deploy.sh

Wait for the upload to complete and update your lambda function. You may follow instructions from this blog as a reference in case you want to create a different lambda function with different package requirements.

Lambda Function-1 (Frame Extractor)-- Follow all the instructions as above but now instead of installing pyTorch install cv2 package and create a new lambda function on AWS and supply its name in the deploy.sh, additionally you may choose to change the filenames too, so as to create a distinction between pyTorch setup and cv2 setup. As for the code to include now you only need to place one python script which is main.py provided in this submission.

Rpi Setup- This pretty straightforward, you only need to transfer the two python scripts provided under the Rpi Setup folder in the submission, you may do this via SSH. See the required setup instructions from the project description pdf provided by the instructor to know how to setup Rpi in general and connect to it via SSH.

5. Individual contributions (optional)

For each team member who wishes to include this project in the MCS project portfolio, provide a one-page description of this team member's individual contribution to the design, implementation, and testing of the project. Different students' individual contributions cannot overlap. One page per student; no more, no less.