CSE 546 — Project2 Report

Qiang Fu, Kiran Shanthappa, Sreshta Chowdary Kampally

1. Problem statement

With the rise of deep learning, we can see more and more machine learning applications in real life, with one of the most impactful applications being facial recognition. With the help of deep neural network models, we can have devices that recognize faces in real time, which could be very helpful in many scenarios, including but not limited to security, authentication and reception work. However, one significant bottleneck in this application is that the model requires high computational power to run, and frequently we may have scenarios where such a device is not available. Cloud computing can solve this problem by moving the computation to the cloud, minimizing the work needed to be done on the local machine. Additionally, using cloud to perform the computation is easily scalable and requires less effort to deploy the service.

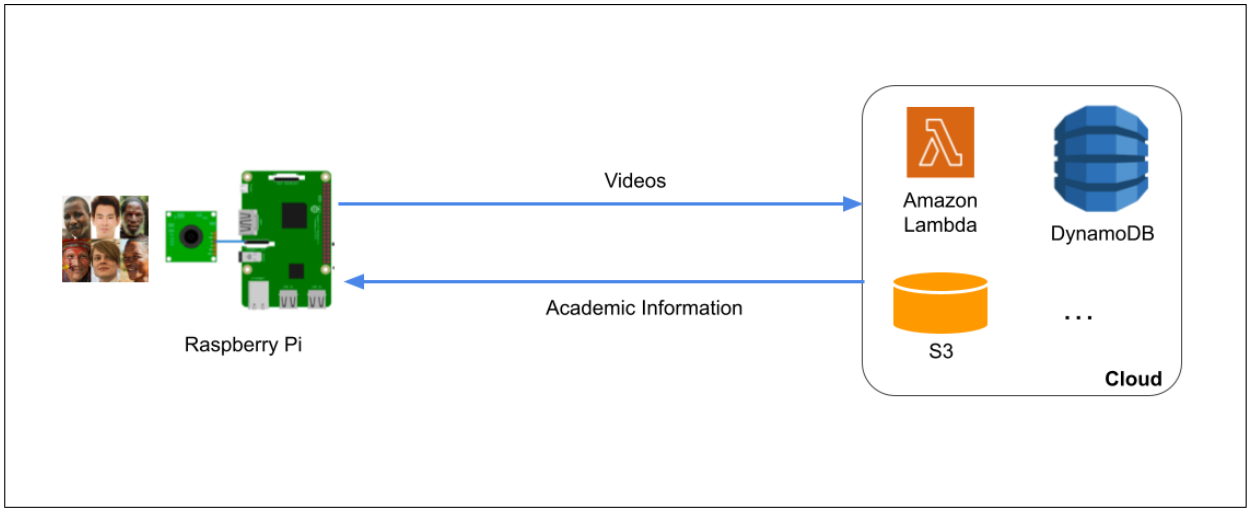
Our team has previously built an IaaS cloud application, capable of receiving requests, waiting for computation to be done on the cloud, then retrieving the results and delivering them back to the users. In addition to that, we built the application in a way that automatically scales up and down according to the amount of requests in the request queue. However, all the functions require us to manually manage entire virtual machine instances, as well as many instance-specific configurations. It is an unwanted burden for the developers.

With the help of Amazon Lambda, a FaaS service, we can reformat our app structure to serverless, and minimize the effort needed to manage the application itself, as well as the auto scaling functionality. In addition to that, we can use Lambda to freely serve multiple edge devices that are submitting requests simultaneously.

Additionally, our previous work was tested only with a set of fake data. In this project, we capture real time videos, and process them to perform face recognition on a model trained with real faces, so the model has some actual real world impact compared to a lab environment.

2. Design and implementation

2.1 Architecture



For this project, we use raspberry pi to take videos with an attached camera and send them to the cloud. These videos will be uploaded to s3, which will invoke our AWS Lambda function to process the uploaded videos. This lambda function extracts the frames from these videos and processes them to recognize faces from the frames. This face recognition result is used to query the DynamoDB table the academic information, and the final query result will then be used to generate an SQS message in a SQS queue, allowing it to be received by our Pi device.

2.2 Concurrency and Latency

In the first project, we deal with user requests within one HTTP POST request, and set the timeout limit to very long(minutes). This form of implementation is not only unscalable, and would have a traffic jam if receiving large amounts of requests simultaneously, with the possibility of overloading and crashing the server. In addition to that, for any application without subprocess, this will block any following code execution until the request finishes and the user gets the result, which is a very bad scenario and we want to avoid that.

In this project, instead of working through HTTP requests, we fully utilize S3, Lambda and SQS to decouple the request creation, data processing and result retrieving. The process starts on the Pi device, where the device captures a video and saves it as a file. Then, it uploads the file to S3, which we have configured to invoke Lambda function on any object put event. The uploading is done in a subprocess, as uploading does not conflict with video recording, this way, we could seamlessly start multiple concurrent requests. Because of the nature of Lambda, it

automatically scales according to usage, therefore we can have concurrent processes dealing with each individual request. Finally, after the lambda function finishes processing, it sends the result as an SQS message. In our Local device, we had another separate script that acts as a message retriever that keeps retrieving and deleting SQS messages. If we run both the retriever and video capture script at the same time, the retriever will keep getting the newest SQS message, therefore continuously updating the newest results. With this structure, we can have almost unlimited(still capped by Lambda) concurrent requests, and since they all should start processing immediately with no conflict, they would have minimum latency during processing.

2.3 Lambda Function

The container image must implement the Lambda Runtime API. The AWS open-source runtime interface clients implement the API. You can add a runtime interface client to your preferred base image to make it compatible with Lambda.

A read-only file system must be supported by the container image. Your function code has access to the 512 MB writable /tmp/ directory. All of the files required to run your function code must be accessible to the default Lambda user. By defining a default Linux user with least-privileged access, Lambda adheres to security best practices. Make sure your application code doesn't rely on files that other Linux users aren't allowed to run. Only Linux-based container images are supported by Lambda.

Add the image to Amazon's ECR repository. Replace 123456789012 with your AWS account ID in the following commands, and set the region value to the region where you want to build the Amazon ECR repository. Authenticate your Amazon ECR registry using the Docker CLI.

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS
--password-stdin 123456789012.dkr.ecr.us-east-1.amazonaws.com
```

Create a repository in Amazon ECR using the create-repository command.

```
aws ecr create-repository --repository-name hello-world
--image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

Tag your image to match your repository name, and deploy the image to Amazon ECR using the docker push command.

```
docker tag  hello-world:latest
123456789012.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

```
docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

Now that your container image resides in the Amazon ECR container registry, you can create and run the Lambda function.

3. Testing and evaluation

For testing, we opened two terminals and ran two scripts at the same time, messages.py and camera.py, to test the application. We attach the piCamera to our Raspberry Pi device, and put our face in front of the camera, so the script captures our face in a short video and uploads it to s3. After that, the cloud part should be able to process the video and send the classification result to an SQS queue. In the other terminal running messages.py, it will keep reading messages from SQS and calculate the latency between video sent and message received, then print them on the screen.

```
pi@raspberrypi:~/project2/cse546-project2 $ python messages.py
Receiving SQS messages...
Empty Queue...
Empty Queue...
Empty Queue...
```

Screenshot of message retriever when no face is detected

```
------------------------------
The 1 person detected With original provided model: Qiang, Computer Science, graduate
The 1 person detected With Online model: Qiang, Computer Science, graduate
------------------------------
Latency: 5.00 seconds.
------------------------------
The 2 person detected With original provided model: Qiang, Computer Science, graduate
The 2 person detected With Online model: Qiang, Computer Science, graduate
------------------------------
Latency: 5.33 seconds.
Empty Queue...
------------------------------
The 3 person detected With original provided model: Qiang, Computer Science, graduate
The 3 person detected With Online model: Qiang, Computer Science, graduate
------------------------------
Latency: 5.45 seconds.
```

Screenshot of message retriever when face is detected, showing prediction done by the two implemented models

We implemented two models for prediction, one is the model provided in the project, the other is from the python face_recognition library available online. We ran both models online which sent the result back in one message so we can differentiate the result.

For the model provided in the project, we have an accuracy around 60%, which is higher than randomly guessing, 33%. However, the model sometimes keeps predicting to one person in the dataset, which is a problem. We tried to increase the training data, add salt to it, and use images directly captured from Pi, as well as changing the optimizer to Adam and increasing the batch size, but they still did not have a good result. It often reaches 100% accuracy on one person, but only occasionally predicts the other person when provided their face. For the external model we used, the accuracy of predicting our face correctly is at almost 100%. We also noticed a minor bug that is easy to fix, but is actually kind of beneficial to the app, when no face is detected, it will break lambda function, and not outputting the sqs message, we decided to keep it like that since it does not impact the main function, and actually helps the app doing its task.

4. Code

camera.py

This is the script used to record videos from pi in .h264 format and upload them to aws s3.

messages.py

This script receives messages from sqs queue in the form of json file which consists of academic information and display the output on to the terminal.

Train_face_recognition.py

This script is provided in this project, used to train the data with a given path and epoch number. It is modified to have a batch-size of 32 and uses Adam optimizer with lr=0.001 instead of the original SGD optimizer.

Dockerfile

The default Lambda user must be able to read all the files required to run your function code. Lambda follows security best practices by defining a default Linux user with least-privileged permissions. Verify that your application code does not rely on files that other Linux users are restricted from running.

In this project directory, add a file named **requirements.txt**. List each required library as a separate line in this file. Use a text editor to create a Dockerfile in your project directory. The following example shows the Dockerfile for the handler that you created in the previous step. Install any dependencies under the ${LAMBDA_TASK_ROOT} directory alongside the function handler to ensure that the Lambda runtime can locate them when the function is invoked. Lambda supports the following container image settings in the Dockerfile:

- ENTRYPOINT – Specifies the absolute path to the entry point of the application.
- CMD – Specifies parameters that you want to pass in with ENTRYPOINT.
- WORKDIR – Specifies the absolute path to the working directory.
- ENV – Specifies an environment variable for the Lambda function.

On your local machine, create a project directory for your new function. Create a directory named app in the project directory, and then add your function handler code to the app directory. The AWS base images provide the following environment variables:

- `LAMBDA_TASK_ROOT=/var/task`
- `LAMBDA_RUNTIME_DIR=/var/runtime`

Install any dependencies under the ${LAMBDA_TASK_ROOT} directory alongside the function handler to ensure that the Lambda runtime can locate them when the function is invoked.

Build your Docker image with the docker build command. Enter a name for the image. The following example names the image hello-world.

```
docker build -t project2 .
```

Start the Docker image with the docker run command. For this example, enter hello-world as the image name.

```
docker run -p 9000:8080 project2
```

(Optional) Test your application locally using the runtime interface emulator. For testing locally on the machine, aws lambda RIC (Runtime Interface Client) is used and the same has been configured in entry.sh script. From a new terminal window, post an event to the following endpoint using a curl command:

```
curl -XPOST "http://localhost:8080/2015-03-31/functions/function/invocations"
-d '{"Records": [{"eventVersion": "2.0","eventSource": "aws:s3","awsRegion":
"us-east-1","eventTime": "1970-01-01T00:00:00.000Z","eventName":
"ObjectCreated:Put","userIdentity": {"principalId":
"EXAMPLE"},"requestParameters": {"sourceIPAddress":
```

"127.0.0.1"},"responseElements": {"x-amz-request-id":
"EXAMPLE123456788","x-amz-id-2":
"EXAMPLE123/5678abcdefghijklambdaisawesome/mnopqrstuvwxyzABCDEFGH"},"s3":
{"s3SchemaVersion": "1.0","configurationId": "testConfigRule","bucket":
{"name": "input-project2","ownerIdentity": {"principalId": "EXAMPLE"},"arn":
"arn:aws:s3:::input-project2"},"object": {"key": "**[KEYNAME ON S3]**","size":
1024,"eTag": "0123456789abcdef0123456789abcdef","sequencer":
"0A1B2C3D4E5F678901"}}}]}'

This command invokes the function running in the container image similar to the S3 trigger for the lambda function and returns a response.

For a new function, you deploy the Python image when you create the function. For an existing function, if you rebuild the container image, you need to redeploy the image by updating the function code.

Handler.py

Handler is the entry point for the lambda function. The handler performs 3 major operation - frame extraction from the video, use the extracted frame to run against the trained face recognition model, and finally the result from the face recognition model is used to get student data. The script does everything after Lambda invocation. First, it reads in a lambda event json, then it attempts to download the file to /tmp/ directory, and extract a frame using ffmpeg. Finally, query the result on DynamoDB and send the final result as an SQS message as a JSON message.

Pictures.py

This script is used to continuously capture photos and save them in a folder with a provided name. It is only used for data collection and is not used in the cloud.

Install and Running with Lambda

With the Dockerfile available, it is very easy to install the application on Lambda. Note that in the following commands, [variable] represents something that may needs to be changed according to AWS account.

First, because the code uses dotenv to load environment variables, a ".env" file needs to be created with values of AWS key, secret key and default region. In addition to that, we need to install awscli and docker with this command:

Sudo apt install awscli docker.io

To login to AWS, we can use this command:

Aws configure

To prepare the AWS environment, we need to create DynamoDB, S3, SQS, ECR with the correct name, as our code currently hardcodes the name of resources. We also need to populate DynamoDB before we could query from it.

ECR can be created with the following command:

aws ecr create-repository --repository-name **[NAME]** --image-scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE

This command also may needs to be ran to set up ecr credentials:

aws ecr get-login-password --region **[us-east-1]** | docker login --username AWS --password-stdin **[numeric ID]**.dkr.ecr.us-east-1.amazonaws.com

In the project directory, run the following commands to build the docker image:

docker build -t project .

Then, push the image to ecr with the following command:

docker tag **[NAME]**:latest **[ID]**.dkr.ecr.**[us-east-1]**.amazonaws.com/**[NAME]**:latest

docker push **[ID]**.dkr.ecr.**[us-east-1]**.amazonaws.com/**[NAME]**:latest

Finally, we can create the Lambda function on AWS console manually using the docker image pushed to ECR. Then, we can manually add the S3 trigger from the corresponding bucket. After this, the application should be completely installed.

The Lambda function needs permissions to interact with many services, so we may need to create a role/edit the existing role to add permissions to services such as S3, SQS and DynamoDB.

5. Individual contributions (optional)

Qiang Fu (ASUID:1212546281):

For this project, Qiang is responsible for organizing the group, arranging meetings and distributing workload among members. Because the team members have different schedules and capability, Qiang needs to set up plans to meet, discuss and work on the project together, while providing each member an acceptable workload that is relative to their capability and speciality. Qiang is also the main contributor to the overall structure design. Qiang did research about available processes and toolsets, and decided the tools to use, including but not limited to EC2 for docker building and SQS for transmitting messages.

During development, Qiang is responsible for testing and managing the authorizations for both users and Lambda roles. Qiang also managed the github repository for the project. Qiang is also responsible for setting up the Raspberry Pi device, including internet, github authentication, SSH and the camera settings.

During the data collection phase for training the provided model, Qiang is responsible for taking photos and processing them into proper format(png) and size(160x160). The model training part is also done by Qiang.

For the coding part, Qiang provided the initial idea of messages.py, and helped debug the final script. A few other scripts are completely done by Qiang, including:

Change_jpg_to_png.py, the script for image format manipulation

Add_noise_to_png.py, a script for adding noise to images

Pictures.py, a script written for collecting photos from the Pi device camera for model training.

During training the model, the result is frequently unstable, and the model is constantly over or under fitting. Qiang is responsible for the entire training phase and tried various methods to improve the training data as well as the training process. For example, the team collected different images in different time, lighting and emotions, and a few iterations of the model are trained with different input data, as well as the batch size and the optimizer. The final model uses batch size=32 and optimizer of Adam with learning rate of 0.001.

Finally, Qiang has worked on handler.py, the main Lambda function script responsible for the entire image recognition process. Kiran first finished the fundamentals of the script, however the script is not tested. Qiang worked with Kiran to update the script, models, debugging and eventually deploying the script to AWS Lambda using docker and ECR. Qiang is responsible for setting up the remote ECR repository and deploying the built docker image to ECR, then creating the lambda function from the image.

Kiran Haralakatta Shanthappa (ASUID: 1222119418):

This project is the practical application of the PaaS functionality through AWS Lambda Function. Kiran created the EC2 instance based on the AMI provided for the model training. Developed the handler.py for processing the videos posted into the s3 bucket. The handler.py extracted the frames from the video and ran the trained face recognition model on the extracted image. Based on the response from the model, the student data is fetched from the DynamoDB. This student data is then sent to the SQS queue in JSON format. The message is for the raspberry pi to get the face recognition result with the respective student data.

The lambda function requires run handler.py calling other functions and having multiple dependencies to successfully perform face recognition. Hence, Kiran researched on the docker file and ECR image. Lambda function is created using the ECR image and setup to trigger based on the data load to the s3 bucket. Kiran used the Dockerfile provided as part of this project and updated it to install the prerequisite packages and copy the dependencies into the designated directory. The Dockerfile also instructs the entry point and execution of the handler.py.

Kiran used the awslambdaric in the entry.sh for simulating the aws lambda functionality on the local system to test the handler function and result from the trained model. The docker run was passed with the payload to simulate the s3 bucket update. This greatly helped in testing the functionality on EC2 instance and later moved to the lambda through ECR.

Tools and Technologies:

1. **Programming lambda function:** Python
2. **Container:** Docker
3. **AWS Cloud Resources:** S3, DynamoDB, SQS, Lambda, ECR, Cloudwatch, and EC2

Learning Outcome

This project has been a great learning experience in understanding and implementing the fully functional and practical application of lambda function. Helped in understanding scope of PaaS, various lambda functionalities such as read-only file system and can only use /tmp/ for write, lambda function trigger, use of awslambdaric for simulating the lambda on EC2 linux instance, adding dependency components in dockerfile, caching to make process faster and simple lambda function creation with runtime environment setup or complex function with ECR images.

Sreshta Chowdary Kampally (ASUID:1224031900):

In this project, Sreshta was responsible for the setup of S3 bucket that is used to store the videos taken from raspberry pi camera, and an SQS queue that is used to send the output from the lambda function to the raspberry pi terminal. Sreshta also created the DynamoDB table that consists of academic information of all the team members like their name, major and year. Sreshta was also involved in brainstorming through various methods of extracting the output from lambda to raspberry pi.

Sreshta worked on the pseudocode and major parts of the code camera.py. This camera.py code records videos on the raspberry pi camera and saves them in the raspberry pi local. It then uses multi-threading to send these videos to aws S3 bucket and upload them. The videos are captured every 0.3 seconds and are immediately sent to S3. Sreshta also worked on the code messages.py that extracts the output from lambda that is sent into SQS queue. The output from lambda is in JSON format and it consists of student information which is sent to the SQS queue. This code messages.py requests messages from SQS queue and when it receives the message successfully, it deletes the message from the queue. This message is shown as the output on the terminal. This code also calculates latency and prints it.

Sreshta is also involved in writing the code pictures.py. This pictures.py captures images of the team members and saves it on the raspberry pi local. These images are used for training the face recognition model.

Tools used:

- **S3**: Used to store videos from raspberry pi camera
- **SQS**: Used to send output from lambda to raspberry pi
- **DynamoDB**: Contains academic information of the team members
- **picamera**: A python module to capture images and videos from raspberry pi camera

Outcome:

This project helped me learn many tools and their functionalities in AWS. It gave me hands-on experience of PaaS Service and FaaS Service. It allowed me to gain an understanding on the picamera module that is used to control raspberry pi cameras. It also helped me to work and learn about creating docker containers in AWS and to train the face recognition model for better accuracy.