Project 2 (Part 2): PaaS

Summary

The second part involves building an elastic cloud application that can automatically scale on-demand in a cost-effective manner using PaaS resources. The application will be developed with AWS Lambda and other supporting AWS services. As the first and most widely adopted serverless computing service, AWS Lambda will be central to this project. This application will be more advanced than the previous project due to increased experience with cloud programming and the efficiency provided by PaaS. The application will deliver a valuable cloud service to users, with the technologies and techniques acquired serving as a foundation for future projects.

Description

Our complete cloud app will be a video analysis application that uses four Lambda functions to implement a multi-stage pipeline to process videos sent by users.

1. The pipeline starts with a user uploading a video to the input bucket.
2. Stage 1: The *video-splitting* function splits the video into frames and chunks them into the group-of-pictures (GoP) using FFmpeg. It stores this group of pictures in an intermediate stage-1 bucket.
3. Stage 2: The *face-recognition* function extracts the faces in the pictures using a Single Shot MultiBox Detector (SSD) algorithm and uses only the frames that have faces in them for face recognition. It uses a pre-trained CNN model (ResNet-34) for face recognition and outputs the name of the extracted face. The final output is stored in the output bucket.

The architecture of the cloud application is shown in Figure 1. We will use AWS Lambda to implement the functions and AWS S3 to store the data required for the functions.

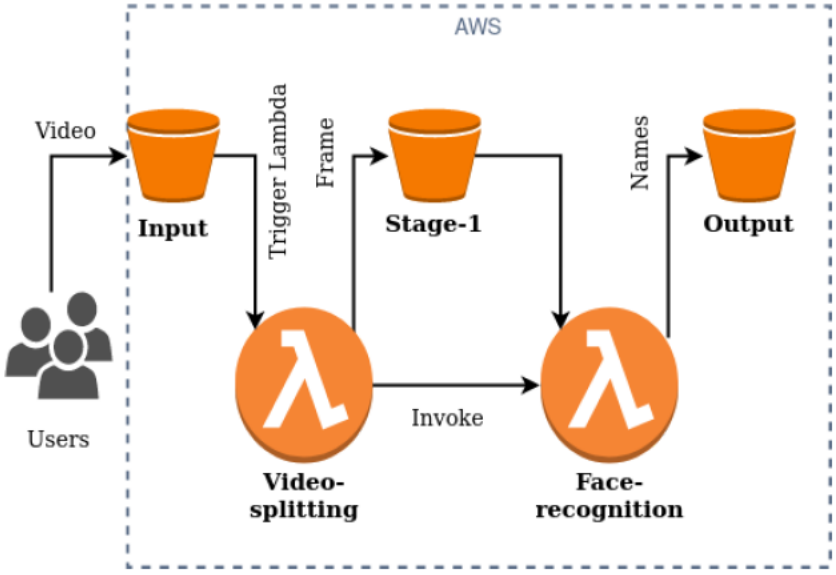

Fig 1: Architecture Diagram of Project 2

Input Bucket

- This bucket stores the videos uploaded by the workload generator.
- The name of the input bucket MUST be **< ID>-input**. Each student submission MUST have only one input bucket with the exact naming. For example, if your ID is "12345678910", your bucket name will be 12345678910-input.
- The input bucket must contain only .mp4 video files sent by the workload generator. (100-video dataset link)
- Each new video upload should trigger an invocation of the **video-splitting** Lambda function.

Stage-1 Bucket

- This bucket stores the result of the video-splitting function.
- The name of the input bucket MUST be **< ID>-stage-1**. Each student submission MUST have only one "**<ID>-stage-1**" bucket with the exact naming.
- The < ID>-stage-1 bucket should have JPG images with the exact name as the input video name with .jpg extension. E.g. If the input bucket has 5 videos with the names test_00.mp4, test_01.mp4, test_02.mp4, test_03.mp4, test_04.mp4, then the stage-1 bucket **MUST** have 5 images with the names exactly as the input video files: test_00.jpg, test_01.jpg, test_02.jpg, test_03.jpg, test_04.jpg.

Output Bucket

- This bucket stores the result of the face-recognition function.
- The name of the output bucket MUST be < ID>-output. Each student submission MUST have only one "<ID>-output" bucket with the exact naming.
- The output bucket should contain text files (.txt) with the same name as the input video with .txt extension. E.g. If the stage-1 bucket has 5 images with the names test_00.jpg, test_01.jpg, test_02.jpg, test_03.jpg, test_04.jpg, then the output bucket should have 5 text files with the names exactly as the original input video files: test_00.txt, test_01.txt, test_02.txt, test_03.txt, test_04.txt.
- The content of each text file is the identified person's name in the image. E.g. test_00.txt should contain only the name of the person identified, i.e. Trump.

Video-splitting Function

- The name of this Lambda function MUST be "**video-splitting**".
- The function is triggered whenever a new video is uploaded to the "**<ID>-input**" bucket.
- The function gets the video file and splits the video into frames using the ffmpeg library.

The function splits the given input in frames using the following command:

```
ffmpeg -i ' +video_filename+ ' -vframes 1 ' + '/tmp/' +outfile
```

Note: To reduce the number of PUT requests, we will extract only one frame from each video.

- The function stores one frame with the same name as the input video in the "**<ID>-stage-1**" bucket. E.g. If the <input_video> is test_00.mp4, then its corresponding frame is stored in a file named test_00.jpg in the "**<ID>-stage-1**" bucket.
- This function invokes the face-recognition function asynchronously as soon as it finishes processing the function.
    - The face-recognition function is invoked with the invocation parameters: "bucket_name" and "image_file_name".
    - The output of this function stored in "stage-1" is passed as the "bucket_name" and the file in which the frame is stored is passed as the "image_file_name" in the invocation parameters.
    - E.g. For a given input test_00.mp4, the face-recognition function invocation parameters would be {"bucket_name":"< ID>-stage-1", "image_file_name":"test_00.jpg"}.
- You can use the provided video-splitting code to implement your Lambda function. **Note**: To conserve our usage of ECR, do not use a container image to deploy this function. You can use Lambda UI to write your function code and add the FFmpeg library externally.

Face-recognition Function

- The name of this Lambda function MUST be "**face-recognition**".
- The face-recognition function is triggered when the preceding video-splitting function finishes processing.
    - This function requires two parameters- "bucket_name" and "image_file_name".
    - E.g. if the previous video-splitting function processed a video "test_00.mp4", then the invocation parameters for the face-recognition function would be "bucket_name" set as "<ID>-stage-1" and "image_file_name" set as "test_00.jpg".
- The face-recognition function processes every image in the following manner.
    - Each image is the frame from the original video which goes through a sequence of OpenCV APIs to detect and extract faces from the image.
    - It computes the face embedding using a ResNet model and compares it with embeddings from the data.pt file.
    - It identifies the closest match. If the match is found, the recognized name is stored as a text file with the same name as the input image, and the file's content is the name of the recognized face.
    - The result of this function is stored in the "< ID>-output" bucket
    - E.g. if the image_file_name in the invocation parameter is test_00.jpg, then the result of the function is stored as test_00.txt in < ID>-output bucket.

- You can refer to the face-recognition-code to implement your Lambda function.
- You can use the provided templates for the Dockerfile and handler code as the starting point for building your container image for deploying this function.

  Note: You **MUST** add the required code/packages in the template code to run and compile your function. You can use the following techniques to reduce the image size in AWS ECR:
  - Use a smaller base image such as python:${VERSION}-slim
  - Use only the required packages and libraries in the Dockerfile and requirements.txt
  - Install Torch and Torchvision for CPU (without CUDA) from the official torch webpage- https://download.pytorch.org/whl/torch_stable.html
  - Save the data.pt in a separate S3 bucket instead of storing it in the container image and download from S3 when needed in the code.