CSE 546 — Project 2 Report

Prasidh Aggarwal, Revanth Suresha, Shriya Srinivasan

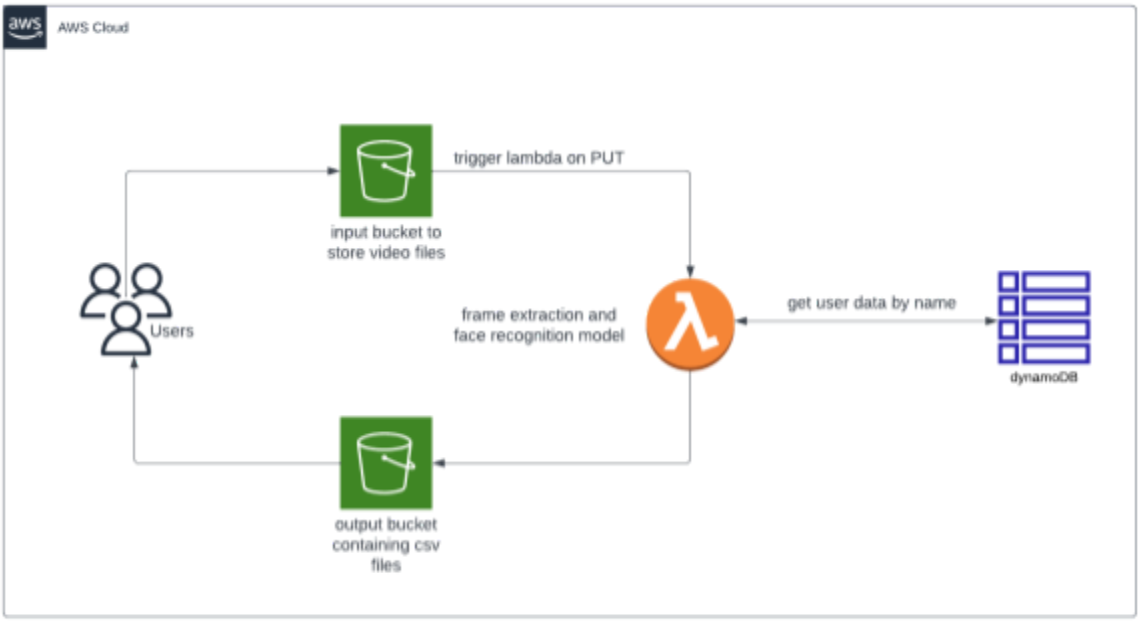
1. Problem statement

The primary objective of this project is to design and implement a scalable, cost-effective, and resilient application utilizing Platform-as-a-Service (PaaS) cloud technology. Our focus will be on developing a sophisticated application that leverages the inherent capabilities of PaaS for ease of development, deployment, and management. The chosen platform for this project is AWS Lambda, a leading serverless computing service, along with other supporting services from Amazon Web Services (AWS). The application in question aims to provide users with a valuable and efficient cloud service, enabling them to harness the full potential of PaaS cloud technology. By utilizing AWS Lambda, the application will automatically scale out and in on-demand, thus ensuring optimal resource usage and cost-effectiveness. This elastic architecture will allow the application to handle varying workloads while maintaining high availability and performance.

AWS Lambda is a serverless computing service that enables developers to focus on their code without worrying about underlying infrastructure management, such as provisioning, scaling, and patching. This streamlined application development and deployment approach provides numerous benefits, including reduced time to market, lower operational costs, and enhanced flexibility. By leveraging these advantages, the project aims to create an application that addresses users' needs and serves as a learning experience for building sophisticated PaaS applications in the future. This project endeavors to construct a robust and scalable application using AWS Lambda and other supporting services, taking full advantage of PaaS cloud technology. The application will provide users with a valuable service while also serving as an opportunity to learn and hone skills in developing advanced cloud applications. The ultimate goal is to create a versatile and efficient cloud service that offers the benefits of elasticity, scalability, and cost-effectiveness inherent in PaaS-based architectures.

2. Design and implementation

2.1 Architecture



Application Architecture

The major components of the architecture include:

1. Two S3 buckets, one each for video requests (uploaded by the user) and result responses.
2. A Private ECR to store docker images.
3. AWS lambda to create a lambda function triggered whenever the input bucket gets a new video uploaded.
4. AWS lambda trigger to connect the S3 put operation with the lambda function handler.
5. Handler of the lambda function that handles the bucket operations and the face recognition logic.
6. Dynamo DB to store student information.

2.2 Autoscaling

AWS Lambda handles auto-scaling by automatically adjusting the number of concurrent function executions in response to incoming requests. This allows the service to maintain high performance and availability without manual intervention. When a Lambda function is triggered, AWS provisions one or more instances of the function to process the event. As the volume of requests increases, Lambda seamlessly scales the number of instances to accommodate the additional load. Behind the scenes, AWS Lambda manages capacity and resource allocation by utilizing "concurrency." Concurrency refers to the number of simultaneous executions of a

Lambda function. When a function experiences a spike in requests, Lambda increases the concurrency level accordingly, creating new instances to handle the incoming events.

Lambda's auto-scaling capability ensures that the application remains responsive and performant even under high workloads without requiring developers to manage infrastructure scaling manually. This eliminates the need to pre-allocate resources or estimate peak demand, resulting in cost savings and optimized resource utilization. The automatic scaling of AWS Lambda ensures that the application benefits from the elasticity and efficiency of serverless computing, thereby enhancing the overall user experience.

2.3 Member Tasks

Prasidh Aggarwal (paggar10) – Efficiently executed the tasks of setting up the project infrastructure. Created an AWS account and defined IAM users with appropriate permissions, ensuring secure access to the required resources. Configured user roles for the Lambda function, establishing the necessary privileges for seamless operation. Proceeded to set up S3 buckets and DynamoDB, creating a solid data storage and retrieval foundation. A private Elastic Container Registry (ECR) was also established for secure image storage. Successfully loaded the DynamoDB with relevant data and built a customized Docker image containing essential libraries and tools. Deployed the Lambda function using the Docker image to AWS, thus completing assigned responsibilities and enabling the project to progress efficiently.

Shriya Srinivasan (ssrin103) – Wrote a Lambda function to process video files and extract frames, ensuring efficient input data handling. Next, the Python face recognition library was integrated, enabling the accurate identification of individuals in the extracted frames. I then developed code to generate output CSV files containing student information derived from the face recognition process. Throughout the development process, tested the Lambda function locally, validating its functionality and making necessary adjustments to optimize performance. Successfully contributed to the project's progress, delivering a robust solution for processing and analyzing video files.

Revanth Suresha (rbangal5) – Started by configuring the Lambda trigger on the input S3 bucket, enabling automatic execution of the Lambda function upon receiving new files. Next, tested the Lambda function using the provided workload generator, simulating real-world scenarios to evaluate its performance and reliability. After verifying the correctness of the output CSV files, assessed the processing times to ensure all requests were completed within a reasonable timeframe. Identified and resolved many issues while testing the lambda function, such as architecture mismatch and permission issues. Further refined the solution to deliver optimal results. Diligently executed each task and contributed to the project's success, ensuring a robust and efficient system for processing and analyzing input data.

3. Testing and evaluation

The team tested the Lambda function locally to ensure seamless integration with the S3 bucket and the overall pipeline. Initially, we verified the correct uploading of video files to the input S3 bucket and triggered the Lambda function. Once triggered, the handler function processed the video frames and performed face recognition. The team ensured that the output CSV files containing the face recognition results were placed in the designated output bucket. Upon successful local testing, we shifted our focus to the AWS environment, where we encountered and resolved several challenges. During deployment, the team encountered issues with building and pushing the Docker image to the Elastic Container Registry (ECR). Debugging efforts led to resolving these problems, allowing for the successful creation and storage of the Docker image. Subsequently, the team faced architecture mismatch issues between the Docker image and the Lambda function, which were addressed and resolved.

Further challenges arose from Lambda function timeout issues and directory permission problems. We debugged these issues, ensuring the smooth operation of the Lambda function. After resolving all identified challenges, the end-to-end testing of the entire pipeline was completed. The final test results demonstrated the successful processing of 100 requests within an impressive 7-minute timeframe, highlighting the efficiency and effectiveness of the developed solution.

4. Code

4.1 Handler

4.1.1 *handler.py*

This python script is designed to perform face recognition on video files stored in an AWS S3 bucket and generate a CSV file with relevant student information. It uses the boto3 library to interact with AWS services, the face_recognition library for face detection and recognition, pickle for loading pre-trained face encodings, cv2 for video processing, and csv for generating output files. It initializes the S3 and DynamoDB clients, specifies the output bucket and table name, and loads the known face encodings from a pre-saved file. The face_recognition_handler function is triggered when a new video file is uploaded to the S3 bucket. Inside the handler function, the bucket name and key are extracted from the event. The video file is downloaded from the S3 bucket and processed frame by frame using OpenCV. To save time, only every other frame is processed and resized to 1/4 of its original size. Faces are detected, and their encodings are computed using the face_recognition library. Then we compare these encodings against the known face encodings. If a match is found, it retrieves the corresponding student information from the DynamoDB table. A new CSV file is created, and the student's name, major, and year are written as a header row. The CSV file is then uploaded to the specified output S3 bucket. It continues processing frames until a match is found and a CSV file is generated, after which the video processing is stopped.