CSE 546 — Project Report

Vishal Reddy Burri (1225412292)

Aditya Mettu (1225490474)

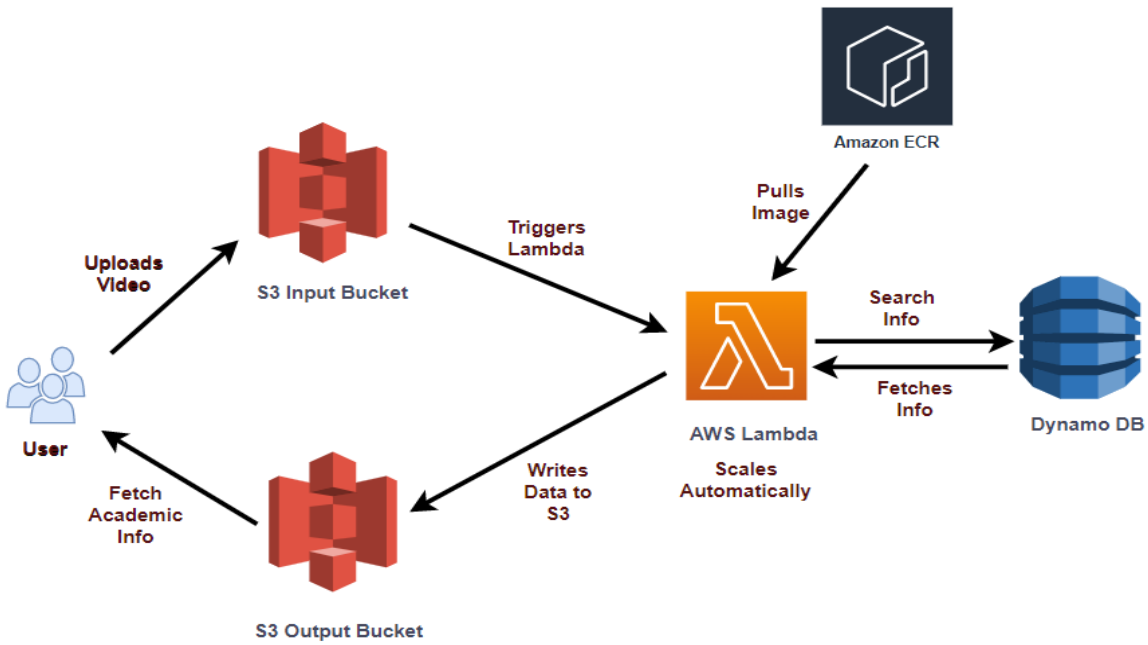
Naga Sreevatsava Macharla (1225467997)

1. Problem statement

For the second project, we will create a flexible and efficient cloud-based application that can automatically adjust its resources to meet demand by utilizing PaaS (Platform as a Service) technology from AWS (Amazon Web Services). This application will be more sophisticated than the previous project and will involve the use of AWS Lambda and other supporting services. The goal is to build a smart classroom assistant that can assist teachers by analyzing classroom videos using face recognition technology, looking up student information in a database, and providing relevant academic information back to the user. PaaS technology will simplify the development process and enable us to build the application entirely in the cloud.

2. Design and implementation

2.1 Architecture



AWS Resources:

- ➢ S3
  - Amazon S3 is a cloud-based object storage service that provides developers and IT teams with secure, durable, and scalable storage for a variety of data types, including images, videos, log files, backups, and data archives. In this application, S3 is used to store input video and output CSV files. Specifically, users upload videos to the input bucket, and the Lambda function stores the resulting academic information as a CSV file in the output bucket. S3 provides high durability, scalability, and availability. It also offers features like lifecycle policies, versioning, cross-region replication, and intelligent tiering that enable cost-effective storage and management of data at scale.
- ➢ Lambda
  - AWS Lambda is a serverless computing service that allows developers to run code without provisioning or managing servers. Lambda functions are event-driven and automatically scaled, which means that they run only when needed and scale automatically to handle incoming traffic. In this application, Lambda is used to process videos uploaded by users, extract frames, perform face recognition, search for academic information in DynamoDB, and store the results in S3. Lambda provides a cost-effective and efficient way to build and run applications at scale. With Lambda, developers do not need to worry about infrastructure management, scaling, or availability, as these aspects are taken care of by the platform.
- ➢ ECR
  - ECR is a fully-managed Docker container registry that makes it easy for developers to store, manage, and deploy Docker container images. In this application, ECR is used to store a customized Docker container image that contains the necessary dependencies, such as FFmpeg and face_recognition libraries, to run the Lambda function. By using ECR, the development team can easily version, tag, and deploy the Docker image across multiple environments and AWS regions.
- ➢ Dynamo
  - DynamoDB is a fully-managed NoSQL database service that provides fast and predictable performance with seamless scalability. In this application, DynamoDB is used to store academic information about known faces. Specifically, the Lambda function searches DynamoDB for the student's academic information based on the name of the first recognized face and returns it as a CSV file. DynamoDB is a highly available, durable, and scalable database service that can handle a large volume of reads and writes at low latency. DynamoDB also provides features like auto-scaling, encryption, backups, and global tables that enable developers to build highly available and scalable applications.

The first step of the application flow involves the users uploading videos to the input bucket in S3. S3 is a simple storage service provided by AWS that allows users to store and retrieve data from anywhere on the web. When a new video is uploaded to the input bucket, it triggers the

Lambda function to process the video. Lambda is a serverless computing service provided by AWS that allows the application to run code in response to events such as the upload of a new video to the S3 bucket. The event is sent as a function parameter to the main lambda function, it then retrieves the bucket name and key used in the event and downloads the video to a local temporary location.

Then the execution goes to the frame extractor module where it first extracts frames from the video using the multimedia framework FFmpeg, which is preinstalled in a custom container image. Ffmpeg is a popular tool for handling multimedia data and is widely used for video processing and encoding. All the extracted frames are sent to the face recognition module which uses the face_recognition library to identify the first face detected in the frames.

Once the name of the recognized face is found it is sent to the student_info module to search for the person's academic information in DynamoDB, which is a NoSQL database provided by AWS. If the academic information is found, the Lambda function stores it as a CSV file in the output bucket in S3. The name of the CSV file is the same as the video name, and the content of the file includes the name of the student, their major, and their year. This allows the user to access the student's academic information in the output bucket after the video has been processed by the application.

2.2 Autoscaling

The number of function instances in Amazon Lambda is automatically scaled to match the volume of incoming requests. In response to incoming requests, more instances of the function are provisioned. Amazon Lambda automatically generates a function instance to handle the request when a function is activated. Amazon Lambda will continue to construct new instances of the function to handle the growing demand if there are numerous simultaneous requests. Amazon Lambda uses this data to dynamically alter the number of function instances to fit demand by tracking the rate of incoming requests and the resources needed to perform each request.

Amazon Lambda will automatically raise the number of instances to manage the additional demand when the concurrency or requests per second metrics surpass a predetermined threshold. Amazon Lambda will automatically lower the number of instances if the load drops in order to maximize resource usage. This process is called reactive scaling and it also has proactive scaling which entails foreseeing future demand based on past consumption trends. AWS Lambda's Provisioned Concurrency functionality enables programmers to pre-warm function instances before they are actually required. Developers may prevent cold starts and make sure a function is always prepared to serve incoming requests by pre-warming function instances.

2.3 Member Tasks

Vishal Reddy Burri (1225412292)

Implemented three modules in this project - Frame Extractor, Face Detector, and S3 Client. These are responsible for extracting frames from a video file, detecting faces in the extracted frames, and interacting with the AWS S3 bucket, respectively. The Frame Extractor downloads the video file from the S3 bucket, stores it in a temporary location, and uses the FFmpeg library to extract JPEG frames from the video. The Face Detector module detects faces in the frames and matches them with known encodings. The S3 Client module has utility methods to download and upload files from/to the S3 bucket. The Dockerfile was modified to copy these modules to the root directory. Unit tests were performed on each module, and end-to-end testing was conducted to verify the lambda's scalability and data integrity in the S3 bucket. The multithreaded workload variant was implemented and used for testing, and lambda concurrency was verified to match the available concurrency in AWS.

Naga Sreevatsava Macharla (1225467997)

Implemented Student Info module to extract specific student information from a DynamoDB table using the name as the partition key. The Handler module was developed as the coordinator for all other modules, integrating the Frame Extractor and Face Detector modules to extract frames from a video file, detect faces, and match them with known encodings. The matched student name was then used to retrieve additional information from the DynamoDB table, and a CSV file was generated and uploaded to an S3 output bucket. The DynamoDB table was set up, and a script was written to insert data from the provided student_data.json file. Unit tests were conducted on each module separately, and end-to-end testing was performed to ensure that all requests were processed within the given time limit and that the data was correctly inserted into the DynamoDB table.

Aditya Mettu (1225490474)

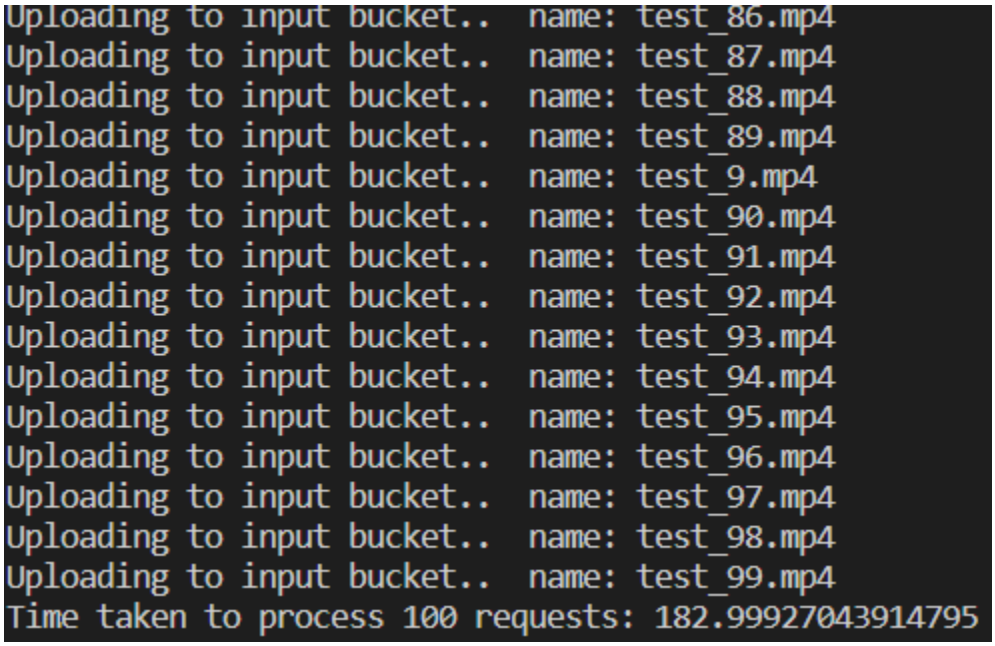
Responsible for infrastructure setup for this project. An Input bucket and an Output bucket were set up, along with IAM roles, users, and access policies to ensure secure access control. An S3 event notification was set up to trigger the Lambda function automatically when new data was available in the Input bucket. A private ECR repository was created to store the Docker container image, which was built locally using the Dockerfile and pushed to the ECR repository. The Lambda function was configured to use the ECR container image to execute the code, with the time and memory limit configurations set as per requirements. End-to-end testing was conducted with the team to verify the system's functionality, with output files in S3 matching the expected output files provided.

3. Testing and evaluation

The lambda function was first tested locally using a customized event as input to ensure its proper functioning. Once the function was verified, an image was generated and uploaded to the ECR. Subsequently, the lambda function retrieves the uploaded image from the ECR when triggered.

We followed the below steps to test & evaluate our application by uploading video files to the input s3 bucket.

1. Triggered workload generator script with both test cases (Test case #1 with 8 videos and Test case #2 with 100 videos).
2. Verified all the videos are uploaded to the s3 input bucket
3. Verified Lambda functions are getting invoked and automatically scaled based on the requests.
4. Ensured the given student data JSON file is correctly inserted into a dynamo db table.
5. Verified number of output files in the s3 output bucket equals the s3 input bucket.
6. Cross-checked the output files in s3 with the provided expected output files and matched.
7. Ensured all the requests are completed within the specified time limit.



By sequentially uploading 100 video files to the s3 input bucket, the time taken to process all the requests is around 3 min with 5 concurrent executions.

With a parallel multithreaded generator I uploaded 100 video files to the s3 input bucket and it took around 80 sec to process all requests with 10 concurrent executions (default reserved concurrency value for new AWS accounts).

4. Code

- *Dockerfile*
    - Modified the given initial code to copy over the modules required to build the image. It contains various steps to install dependencies and copy the code to the function dir for the image to work.

- *handler.py*
    - It contains the lambda function code which is triggered on the s3 upload. face_recognition_handler method is the starting point of the trigger which takes the event as a parameter and classifies the first recognized image.
- *face_detector.py*
    - This module is responsible for detecting the first available face encoding among the given frames. It uses the face_recognition library to compare the faces with the provided encodings and returns the name of the face recognized.
- *frame_extractor.py*
    - This module is responsible for downloading the video file from the s3 bucket to a temporary location and extracting frames using the FFmpeg library. It returns the list of all frames extracted from the video.
- *s3_client.py*
    - Contains utility functions to upload files to s3 and download files from s3.
- *student_info.py*
    - This module fetches all the details (name, major, year) based on the name key from the dynamo database. It returns comma-separated values to the caller to later store it as a CSV file.
- *encoding*
    - This provided file contains the encodings of the known faces
- *entry.sh*
    - This provided script sets up the necessary environment and executes the python code for a Lambda function, either locally or in the AWS Lambda environment, depending on whether the AWS_LAMBDA_RUNTIME_API environment variable is set or not.
- *requirements.txt*
    - This provided file contains all the required dependencies to install for the image to run successfully.

Steps to install and execute:

1. Install docker in your environment
2. Build the image using the Dockerfile provided
    a. docker build -t cse546-proj2 .
3. Create a Amazon ECR repository and push the image.
4. Configure AWS lambda with the image pushed to ECR and add a trigger on s3 object-create events on the input bucket.
5. Test the outputs using the provided workload.py script
    a. python workload.py

Individual Contributions:

Vishal Reddy Burri (1225412292)

Design:

As part of the design phase, I collaborated with my team and discussed the reference architecture given in the project description. We analyzed the requirements and decided to use the given architecture as it is scalable and separated the tasks into different modules so that it is easily integrated and tested later once each of the team members completes their tasks.

Implementation:

- Implemented Frame Extractor Module:
    - The Frame Extractor module is responsible for extracting frames from a video file. It takes in two parameters, the bucket_name, and object_key which are used to download the video file from the S3 bucket. The downloaded file is stored in a temporary location on the local machine. The module uses the FFmpeg library to extract JPEG frames from the video in the temporary location. Once the frames are extracted, they are returned to the caller.
- Implemented Face Detector Module:
    - The Face Detector module is responsible for detecting faces in the extracted frames. It takes in a list of frames that were returned by the Frame Extractor module as input. For each frame in the list, the module uses the face_recognition library to load the frame and check if there are any face locations in it. If there are any face locations, the module matches the faces with the known encodings and returns the name of the matched encoding to the caller. Subsequent frames are ignored.
- Implemented S3 Client Module:
    - The S3 Client module is responsible for interacting with the AWS S3 bucket. It contains two utility methods, the first one is responsible for downloading a given file from the S3 bucket. It takes in the bucket_name, object_key, and local_file_path as parameters and downloads the file from the S3 bucket to the local machine's file system at the specified local_file_path. The second utility method is responsible for uploading a given file to the S3 bucket. It takes in the bucket_name, object_key, and local_file_path as parameters and uploads the file from the local machine's file system to the S3 bucket.
- Made changes to the Dockerfile to copy the above modules to the root directory.

Testing:

During testing, I performed unit tests on each module to ensure their individual functionalities and conducted end-to-end testing after all the modules are integrated into the handler module. We verified that the lambda was able to automatically scale to meet the demand, and ensured S3 bucket objects data matched with the given correct classification file. We have also implemented and tested the multithreaded workload variant and checked the lambda concurrency matches the default available concurrency in AWS.

Individual Contributions:
Aditya Mettu (1225490474)

Design:
As part of the design process, my team and I collaborated to discuss the reference architecture provided in the project description. I discussed with my team the major components required for the project and was involved in setting up the required infrastructure.

Implementation:
- Set up the S3 buckets
    - In this part of the project, two S3 buckets were created: an Input bucket and an Output bucket. The Input bucket is used to store the video files that need to be processed by the system, while the Output bucket is used to store the processed data.
- Set up IAM Roles, Users, and Access
    - To ensure secure access control, IAM roles, users, and access policies were created. Access policies were created to define specific permissions for various components in the system.
- Setup Trigger Point from S3
    - An S3 event notification was set up to trigger the Lambda function whenever a new video file is added to the Input bucket. This ensures that the Lambda function is automatically invoked whenever new data is available in the Input bucket.
- Setup ECR
    - A private ECR repository was created to store the Docker container image that was used to execute the Lambda function.
- Setup Docker & Build Image
    - Docker was first installed locally and used the given Dockerfile to build a container image of the Lambda function code. The image was then pushed to the above ECR repository so that it could be used to execute the Lambda function.
- Setup Lambda
    - A Lambda function was created to process the video files stored in the Input bucket. The Lambda function was configured to use the ECR container image to execute the code. Also, modified the time limit and memory limit configurations to run this image.

Testing:
As part of the testing phase, I conducted unit tests on each module to verify their individual functionalities. Later, I performed end-to-end testing with my team after integrating all the modules into the handler module. During this phase, I verified that the number of output files in the S3 output bucket matched the number of files in the S3 input bucket. Furthermore, I cross-checked the output files in S3 with the expected output files provided and confirmed that they matched, ensuring the system was functioning as intended and meeting the project requirements.

Individual Contributions:
Naga Sreevatsava Macharla (1225467997)

Design:
During the design phase, I worked with my team to analyze the project requirements and consider the reference architecture provided in the project description. After careful consideration, we decided to use the provided architecture as it was scalable and allowed us to separate the various tasks into distinct modules. This approach made it easier for each team member to complete their individual tasks and for us to later integrate and test the different modules as a cohesive system.

Implementation:
- Implemented Student Info Module:
    - The Student Info module is responsible for extracting particular student info from the dynamo database. It takes in a parameter name which is used as a partition key for a table. The modules query the student_info table with the key as the name. Once the query is executed, it retrieves all the information about the student from the table. This information is then formatted in a comma-separated way and returned to the caller.
- Implemented Handler Module:
    - The Handler module acts as a coordinator for all other modules and manages their integration. It first receives an event trigger and then calls the Frame Extractor module to extract frames from a video file stored in an S3 bucket. These frames are then passed to the Face Detector module to detect any faces and match them with known encodings. The matched student name is then sent to the Student Info module to retrieve additional information about the student from a DynamoDB table. Finally, the Handler module generates a CSV file containing the gathered information and uploads it to an S3 output bucket.
- Setup Dynamo DB:
    - I set up a DynamoDB table called 'student_info' in the AWS Dynamo database. The partition key of the table was set to 'name', which will make it easier to query student information using the name as the search parameter. To insert the data into the table, I used the provided 'student_data.json' file and wrote a script to automate the insertion process. The script read the data from the JSON file and used the AWS SDK to insert the data into the 'student_info' table.
- The Dockerfile was modified to include copying of the previously mentioned modules to the root directory.

Testing:
During the testing phase, I conducted unit tests on each module separately to verify their individual functionality. Subsequently, I performed end-to-end testing by integrating all modules into the Handler module. During this testing, I ensured that all requests were processed within the given time limit and that the given student data JSON file was correctly inserted into the DynamoDB table.