CSE 546 — Project1 Report Group Alpha

Abhiram Gulanikar (1222295360) *Prathmesh Sambrekar (1222154895)* *Vinay Pandhariwal (1219501465)*

1. Problem statement

Develop an elastic application that can automatically scale out and in on-demand and cost-effectively by using the IaaS cloud. The app will provide an image recognition service to users, by using cloud resources to perform deep learning on images provided by the users.

Requirements:

A. The application should be able to deal with multiple requests without dropping any request.

B. It should automatically scale out when the request demand increases, and automatically scale in when the demand drops.

C. The system should be able to store the input image and output classifications in the input and output S3 buckets.

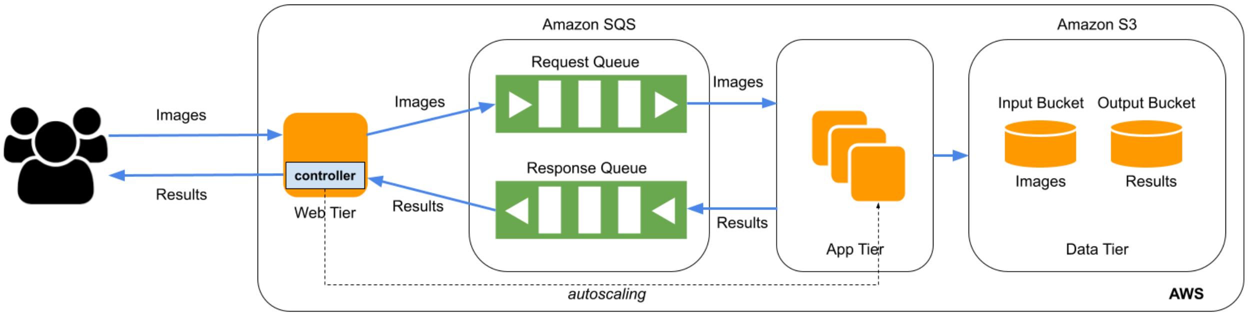
D. AWS AMI with the deep learning model: ami-0bb1040fdb5a076bc

E. AWS Region: us-east-1

2. Design and implementation

2.1 Architecture

The following is the overall architecture diagram for the project.



(Fig 1: Architecture)

2.1.1 Components

a. Web Tier

A tomcat server that receives (listens) incoming post requests from users to upload photos makes up the web-tier. The server keeps the photos in SQS queues as input and puts them in S3 buckets as input for app-tier processing. Web-tier scales out the app-tier instances (maximum of 19) for quicker processing of the requests/images when the number of pictures (msgs) in the sqs queue rises. In order to receive the results of image recognition, the Web-tier also polls the SQS queue (output) and provides the appropriate response. Each request is processed by the web-tier using multithreading, which spawns new threads and creates new instances of the app-tier.

b. App Tier

Image recognition in images is the responsibility of the App Tier. The web tier initiates the app tier to handle the input queue's images. As soon as the App Tier is launched, it polls a name from the input queue for an image. Additionally, the actual image is downloaded from the input S3 bucket. The face recognition python script is then invoked for the image to receive the categorization, and the face recognition task is completed. Once the classification is complete, the output S3 bucket stores the name of the input image and the classification result as a key value pair. Additionally, the output queue receives the input image and categorization for consumption. Finally, the app tier performs scaling down if it discovers any further photos in the queue. If not, it repeats the process above.

c. AWS EC2

The App Tier and Web Tier of the application are hosted by the EC2 instance. The app-tier is in charge of processing the queued image names and download the corresponding images form input bucket at S3 to produce the recognition results for the images. Additionally, in the context of the web tier, it is accountable for listening to incoming classification request and sending it to the input queue, processing the request, managing app tier scaling, and returning the classification output by polling the response queue. The main service for this application and functionality is provided by this AWS service.

d. AWS SQS

A distributed queue offered by AWS services is called SQS, or Simple Queue Service. We are employing the FIFO SQS queue for this project. The order is upheld and a single request is processed by the FIFO SQS queue. In this project, we're using two FIFO SQS queues. The input queue, or queue of input (request) images, is the first. The Web Tier populates this. The classification result is stored in the second queue which is the response/output queue and is consumed by the web layer after being filled by the app tier.

e. AWS S3

The AWS S3 or Simple Storage Service is a key-value store. It is used at two places in the project. The first S3 bucket (input bucket) is responsible for holding the image file name and the actual file. This is the place from where the app tier downloads the image for classification. Finally, the second S3 bucket (output bucket) holds the image name and classification result pair.

2.2 Autoscaling

Based on the requests available in the input queue the web tier automatically scales up and down providing the autoscaling functionality. Based on the messages queuing up in the input SQS queue, the web-tier determines how many app-tier instances to generate. The application continuously tracks both the quantity of active app instances and the quantity of requests in the input queue in the web layer. The web tier automatically starts the app instances in response to demand when the number of instances is fewer than the number of app instances. Web-tier spawns app-tier instances equal to the messages present in the queue if the number of messages present in the queue is less than 19. The total number of requests are divided by the number of app instances running and those many number of threads are created to process that many requests and also spawns maximum app-tier instances it can create (19) when the number of input requests in the input queue exceeds the maximum count of app instances, which is 19. For auto-downscaling, after processing the requests and providing the recognition result the app-tier instance polls the input queue for 60 seconds before getting terminated if no input requests are present in the input queue. This is how the downscaling logic works.

2.3 Member Tasks

Abhiram Gulanikar

- **Design and implementation:**
  I was responsible for designing and developing functionalities for the App-tier and the web-tier. I created the logic flow in a function to arrange and order the functionalities such as continuously poll the input queue and read the input image id, download the corresponding image file from S3, run the deep learning algorithm and push the results in the output S3 bucket and the response queue. I also implemented the actual logic to insert the output in S3 and and the response queue. I further implemented the logic to terminate the EC2 instance after polling the input queue for 60 secs if no new message was found in the queue that is scaling down. I implemented the logic for EC2 instance creation and also contributed for autoscaling logic in the web-tier. We had to set up the necessary resources on the AWS portal in order to construct the web tier application modules, including creating the security ID and key for the AWS account, creating the EC2 instance for the web tier using the provided ami id, key-value pair, etc. I have created web tier modules so that they use a single constant file to construct all other AWS resources, such as AWS SQS input and output, AWS S3 input bucket and output bucket, and EC2 instances using the AWS SDK for Java.
- **Testing and debugging:**
  I performed testing for each step discussed above i.e. to verify if messages are getting consumed from the input queue, check if output is getting stored in S3 then to check if the classification result is being retrieved and finally to check for app instance termination. I also debugged the above functions by running them on an EC2 instance.

Prathmesh Sambrekar

- **Design and Implementation:**

  I worked on the App-tier and implemented it in Python and few functionalities in the web-tier side. I was responsible for implementing the logic to create the output bucket, the output response queue, fetch the input queue length, the logic to read from the input queue and get the image id. I used boto3 from python to implement the above functions. I also created a logic to check if the output queue and the bucket already exists if not create them. I also created a python file containing all the constants such as AWS region, the credentials etc to easily fetch them from other parts of the code.

  Further, I worked on the autoscaling logic as well. I had to take into account several limitations when creating the auto-scaling capability for the web-tier application. First of all, we must use an AWS free-tier account, which limits our ability to create EC2 instances to 20. I can scale the app-tier to a maximum of 19 instances while the web-tier is operating on an EC2 instance. As a result, I based the autoscaling design on the aforementioned parameter. Second, we must plan the auto-scaling component such that it may scale in and out as needed based on the input photos we receive. For instance, if a user sends 50 photos, it should start 19 app-tier instances that would handle the request concurrently. Additionally, it should only launch 5 app-tier instances if a user delivers 5 photos. I used multithreading to accomplish auto-scaling since the web-tier manages a variety of functions for the application, including resource creation, incoming request management, polling the output queue for categorization results, and more.
- **Testing and debugging:**

  I have tested the app-tier instance auto-scaling using a variety of different input requests. To demonstrate certain test scenarios, I used five input photos to test the application. Additionally, I tested the program by sending 5 requests at once, each with a single input image. I also tested the application with 100 concurrent requests, each with an input image, to determine its maximum scaling capacity. I also contributed in debugging the app-tier code and web-tier code and made sure that the message sent from the app-tier in the output queue is parsed in same format when the web-tier polls the output queue for the classification result.

Vinay Pandhariwal

- **Design and Implementation:**

  We looked into various implementation strategies for the application's essential functionalities and how they would fit into the architecture before settling on the standard spring boot API design to construct the modules. The project is divided into two main sections (web tier and app tier). The main characteristics of the entire project were the web tier's numerous separate modules, of which I contributed to the design of Server and API Creation, put Images to S3 and initialize EC2 instances, adding image name to AWS SQS input, polling output queue for recognition results etc. The following activities are carried out by the RESTful Web Services for the backend that I created in spring boot using various modules from the application. Implemented a single repository that uploads all photos acquired from the client-side workload generator to an AWS S3 Input Bucket. Added a general service that modifies file formats and a technique for adding all uploaded image names to the SQS Input queue. Put the key as an image and the value as an output in the AWS S3 output bucket and the AWS SQS output queue. Delete the obtained message from the output queue once the output answer has been retrieved from the API. I also used multi-threading to upload each image to an S3 input bucket and an AWS SQS queue. The official documentation from AWS was very helpful in implementing the above.

- **Testing and debugging:**

  The web tier was rigorously tested beginning with the project setup and ending with the verification of every intermediate result. I've added one console output for testing purposes to confirm that photos are uploaded to the S3 bucket and also queued up to the SQS queue. EC2 instances will launch in accordance with the requirements after the images are submitted to the input AWS SQS. Following completion of the procedure, I checked the key and value for each image in the Output S3 Bucket and compared it to the desired results. Additionally, I have also participated in integration testing of the entire program, and we have repeatedly tested our own application to ensure that it is operating and producing results.

3. Testing and Evaluation

1. **Test Case:** When new requests are received, new app-tier instances are produced.

   Evaluation: On the EC2 instances portal, app-tier instances are created and verified.

2. **Test Case:** New requests are received and are inserted in input SQS queue.

   Evaluation: The amount of messages in the SQS input queue corresponds to the total number of requests.

3. **Test Case:** Images provided as request body attachments are kept in the input S3 bucket.

   Evaluation: Images are downloaded from the S3 input bucket and saved there before being reviewed.

4. **Test Case:** When the volume of requests grows, scale up app-tier instances.

   Evaluation: Depending on the number of messages in the input SQS queue, the app-tier scales out. If five photos are uploaded at once, five instances of the app-tier were formed. Instances of the app-tier were produced in the maximum number of 19 if 25 photos were uploaded at once.

5. **Test Case:** Enqueued in the output SQS queue is the classification result for images.

   Evaluation: There are 5 messages in the Output SQS queue for 5 requests.

6. **Test Case:** Results of image categorization are kept in an output S3 bucket.

   Evaluation: Checked the output S3 bucket and confirmed the outcome using the data found in the given CSV file.

7. **Test Case:** When there are fewer messages in the SQS input queue than there are active app-tier instances, the app-tier instances are scaled down.
   Evaluation: Waiting app-tier instances get terminated.

8. **Test Case**: Recognition for just 5 image input
   Evaluation: Recognition result for 5 images is displayed.

9. **Test Case**: Recognition for 100 image inputs
   Evaluation: Results for the 100 images are displayed.

4. Code

App Tier: Below is the structure and explanation of the App-tier codebase which is implemented in python

- **Constants.py:** The file contains all the constants like Queue Names, AWS Access key, AWS Secret key, Bucket names etc.
- **App.py:** The file contains the functions which implement the logic for polling the input queue, receiving the message from SQS Input Queue, downloading the image file from input bucket in S3, processing the image and pushing the result in output queue and output S3 bucket, deleting the message from the queue and terminating the App-tier instance once there are no requests in the input queue for 60 seconds.

Web Tier: Below is the structure and explanation of the Web-tier which is implemented in Java

- The main package is **com.aws.CCalpha.project1.webtier** this package contains three packages:
  - **com.aws.CCalpha.project1.webtier.controller**
    com.aws.CCalpha.project1.webtier.s3
    com.aws.CCalpha.project1.webtier.sqs
- The main package **com.aws.CCalpha.project1.webtier** contains the following files:
- **AwsCCAlphaProject1WebtierApplication.java**: This is the main Starting point of the webtier Spring Boot Application. Here we call the main threads of webtier AutoScaleService and ResponseService. Also here we initialize AWS services.
- **AutoScaleService.java**: This Service keeps on running in background to create new ec2 instance, initialize and takes care of naming the instances.
- **AwsConstants.java**: This file contains all the important constants required for JAVA AWS Connection. This file is required as the application.properties is not accessible at bean creation stage.
- **AwsStubs.java**: It holds the ec2 client and the method to initialize the SQS and S3.
- **ResponseService.java**: This is the another main thread whose task is to keep listening to the output queue and provide results back to the calling handle.
- The package **com.aws.CCalpha.project1.webtier.controller** contains the following files
- **AwsRestController.java**: This file exposes the POST mapping to the handler and therefore provides a medium to get the image recognized and classified.

- The package **com.aws.CCalpha.project1.webtier.s3** contains the following files:
- **AwsS3Operations.java**: This file contains the S3client and all the operational methods required for access and handling S3 buckets and image upload.
- The package **com.aws.CCalpha.project1.webtier.sqs** contains the following files:
- **SQSOperations.java**: This file contains the SQS client and also contains the methods which are required for the access of the SQS queues.

STEPS TO RUN THE PROGRAM

1. Create an EC2 Instance from the AMI Image provided holding the image classifier.
2. Install python libraries boto3 and ec2_metadata using the following commads

   pip install boto3

   pip install ec2_metadata
3. Use Secure Copy Protocol to move the Apptier python scripts to the EC2 instance using the command for both App.py and constants.py

   scp -i <key pair file path> <python script path> ubuntu@<AWS EC2 Public IPv4>:~
4. Create a directory Images in the EC2 instance at same location as the python scripts.
5. Create an AMI using this EC2 instance and save the AMI ID.
6. Create another EC2 instance (or reuse the current EC2 instance) for the Webtier application.
7. Update the saved AMI ID, AWS Access Key ID and AWS Secret Key in the AWSConstants file in the Web-tier.

8. Use maven to create a jar of the Webtier application using the following command

   mvn clean install
9. Move the jar file created in the target directory to the EC2 instance using the secure copy protocol as follows

   scp -i <key pair file path> <Webtier application Jar file path> ubuntu@<AWS EC2 Public IPv4>:~
10. Modify the permissions in the EC2 instance for the jar file and make it executable using the command

    chmod +x com.awsCCalpha.project1-0.0.1-SNAPSHOT.jar
11. Start the Java application using the following command after using ssh to the EC2

    Java -jar com.awsCCalpha.project1-0.0.1-SNAPSHOT.jar

12 Send POST request to the WebTier REST Controller as follows through Postman and body will consist of image file with name “myfile”:

http://44.202.237.157:8080/recognizeImage