CSE 546 — Project Report — Group 13

Narasimha Arun Oruganti(1223956669)
Jay Solanki(1219776844)
Tanmay Jena(1223185782)

1. Problem statement

Develop an elastic on-demand application on AWS that scales automatically based on input size to perform image recognition with the help of the given Deep Learning Model.

The application should satisfy the following requirements:

- Ability to deal with multiple requests and process them efficiently without dropping any.
- Ability to scale up and scale down when the number of images increase or decrease accordingly.
- Ability to store the input and the classification results in S3 buckets.

2. Design and implementation

2.1 Architecture

The team has followed the architecture depicted in the following image for developing the application:

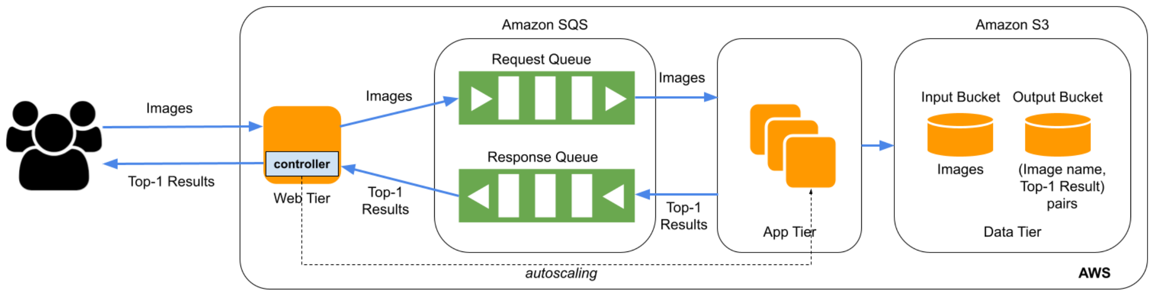

Fig 1. Architecture

2.1.1 Components

a. Web Tier:

This consists of a Node.js server that keeps listening at port 3000 for incoming requests of images. The images that the server listens for are uploaded using the workload_generator. The web tier stores the images into the S3 bucket by converting them into base 64 json format. It also pushes them into the receiver SQS queue after receiving the input as well as listens to the response SQS queue for the classified results of respective images.

b. **App Tier:**
   This tier is responsible for image recognition. It is initiated by the web tier in the form of The App Tier is responsible for image verification. The app tier is started by the web tier to process images from the input/receiver SQS. After fetching the image, it invokes the image_classifier.py to initiate the deep learning model. After the classification is complete, the input image name and the classified value are stored in the S3 output bucket in the form of a key-value pair. Along with that, this key-value pair is passed to the output/response queue for further processing. If the App Tier detects any more images in the input/receiver SQS all these steps are repeated or else it scales down by terminating the instance.

c. **EC2:**
   This AWS is the principal service for the application. EC2 instances are responsible for hosting the app tier and the web tier. The app tier is responsible for processing images from the queue and fetching the classification result. The web tier on the other hand, is responsible for polling the queue to find out incoming requests for classification, scaling, and returning the output classification pair.

d. **S3:**
   The Simple Storage Service(AWS S3) stores objects in the form of a key-value pair. We are using two S3 buckets - input and output. The input bucket stores the input image name and the file. The output S3 bucket stores the image name and its classification result.

e. **SQS:**
   The Simple Service Queue is a distributed queue provided by AWS. Here, the FIFO queue is used for maintaining the order of input images and processing them accordingly. The input FIFO SQS is responsible for the image input and is populated by the web tier. The output queue holds the classification result which is populated by the app tier and consumed by the web tier.

2.2 Autoscaling

Autoscaling of instances happens through the web tier, which acts as a controller. For more requests, autoscaling creates more app-tier instances.
Autoscaling focuses on the number of requests in the input SQS and the existing instances that are either running or stopped. The web-tier keeps track of the size of the input/request SQS in order to scale up or down. Since there is a limit on the maximum number of app-tier instances(20), the number of valid requests are evaluated by ceil(Input SQS Size / 3). Autoscaling then was done using aws-sdk’s runInstances, startInstances and stopInstances APIs.

2.3 Member Tasks

a. **Tanmay Jena(1223185782):**
I was responsible for the initial setup of the web tier in the AWS. I used Node.js to set up the post route where we would receive the requests from the workload generator. I created the input and output buckets that will contain the input images and the classification results respectively. I also handled the code for writing into the s3 buckets, and tested them thoroughly to verify that we got the correct output. I was also involved in auto scaling of ec2 instances where we established the metrics and scenarios to scale up or scale down the application. I also made sure that the response object from the post route was correct and it was being displayed correctly in the console.

b. **Narasimha Arun Oruganti(1223956669):**
I was responsible for writing the cron job that would start the application every time a new instance was created. I went through the Linux commands that were used to perform the actions. I was also responsible for creating the child process to execute the wait for the image classification in the python module. Since Node.js doesn't support multithreading, I made use of Promises chaining to await for classification to get over and then send it to the SQS response queue. I created the process that would create and delete EC2 instances via Node.js. I was also involved in auto scaling of ec2 instances where we established the metrics and scenarios to scale up or scale down the application.

c. **Jay Solanki(1219776844):**
I was responsible for connecting to the SQS queues that we created in the EC2 console. I set up the queues for continuous polling using an NPM package called sqs-consumer. This made sure that the application was continuously processing requests if there are any in the queue. I also handled the application-tier logic and the error handling as well as we were frequently running into issues. Error handling made sure we minimized errors during the development process. I also made sure that the python classifier was correctly acquiring images from the app tier and providing the correct output. Finally, I performed the end to end testing of the application and made sure that we were not dropping any requests during auto-scaling and the classification result was preserved in the S3 bucket.

3. Testing and evaluation

- For testing the application, the first that we did was to run the web-tier. Sending in user requests and enqueuing them in the input/receiver SQS.
  The evaluation was that input SQS contains messages equal to the number of requests.
- Images sent need to be stored in the input S3 bucket.
  Images are successfully stored and can be verified by downloading.

- Storing image classification results in the output S3 bucket.
  Verifying the output S3 bucket results by tallying with the given CSV file iif the image is classified correctly.
- Image classification result for 1 image in a request was displayed correctly.
- Image classification result for 3 images in a request was displayed correctly.
- Image classification result for 100 images in a request was displayed correctly.
- Extensive unit testing was undertaken to verify working of each function and file.
- Integration testing was undertaken to verify end-to-end working of the application.

4. Code

4.1 Code base:

For the web tier application, below is the list of methods:

- **app.js :** This is the entry point of the web tier application. The app goes live and listens for HTTP requests.
- **app.post(/):** This is endpoint listening for post requests which are being sent by the workload generator. The payload consists of the image which needs to be classified.
- **Create_ec2_instance** : This method is used to initialize an ec2 instance. It contains the configurations needed and awaits for the instance to be created.
- **Start_instance** : This method checks if there are enough permissions to start the ec2 instance and starts it, else throws an error.
- **Stop_instance** : This method checks if there are enough permissions to stop the ec2 instance and stop it and all the child processes, else throws an error.
- **Consumer** : Uses a npm package to connect to the SQS queue and continuous polling.

Explain in detail how to install your programs and how to run them.

For the App Tier application, the following methods are used:

- **Read.js** - is the entry point for the app tier application. The app continuously polls the Request queue and reads the images from the request and saves on the PC.
- **Consumer Queue**: In the handlemessage part of the queue we read the images and spawn the image_classification.py function and classify the image.
- **SendTheResponse**: Once the image is classified, we send the classification result to the response queue as well as the S3 output bucket. Also, we send the input images to the S3 input bucket.
- **S3**: This is used to configure the S3 buckets using aws-sdk, and connect and send data to the buckets.

Cron job: This is used to start the app.js in the new instances of EC2. So whenever a new instance is created, it runs automatically to start the application.

4.2 App Setup:

1. Connect to the EC2 instance of the web-tier AMI and make sure it is running.
2. In the local terminal use the ssh command given in README to connect to the web tier.
3. Navigate to the /home/ubuntu/nodejs_server/cloudcomputing-nodejs-server/app.js and run the app.js. The web-tier is ready to receive the requests from the local PC.
4. Launch the workload generator from the local PC and send the request to the http://{web-tier-ip}:3000/
5. Use the python commands to send the request as per workload generator.