CSE 546 — Project1 Report

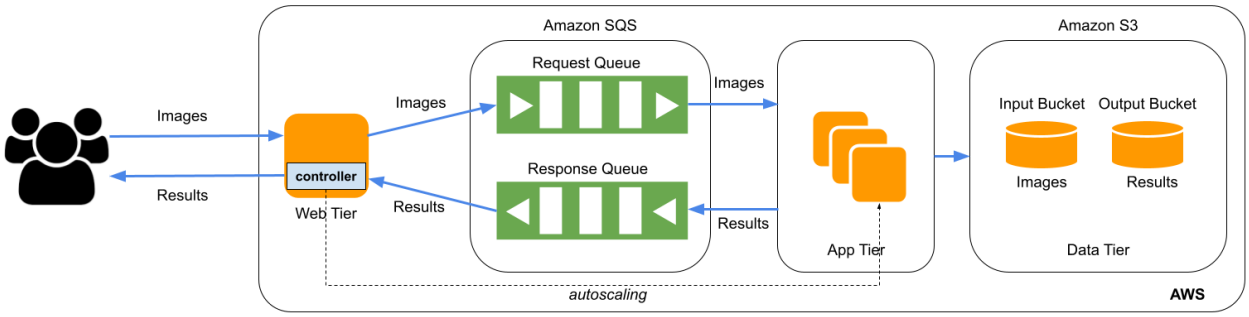
Sudarshan Reddy Mallipalli - *Darshin Shah* - *Venkata Pavana Kaushik Nandula*

1. Problem statement

Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimum management effort or service provider interaction. With the 6 essential characteristics of cloud computing, businesses have many advantages like minimizing the cost and effort, improving the speed and availability of data, improving the reliability and backups. This project "Face Recognition as a Service" is an end-to-end application that is developed to provide the recognition of images by using several features that are present in the Amazon Web Services. Different Amazon Web Services(AWS) such as **EC2** (Elastic Compute Cloud), **SQS** (Simple Queue Service) and **S3 bucket** (Simple Storage Bucket) are used to generate an application that takes the images as input on the web tier and classifies the images using the deep learning model on the app tier and results are saved on the S3 bucket. This application is developed such that instances at the app tier are automatically scaled-in and scaled-out depending upon the number of the images getting passed from the request queue.

2. Design and implementation

2.1 Architecture



1. **Web Tier:**
   Web Tier is implemented using the EC2 instance which holds the code which continuously runs to process the images from the user. The Flask server is used to create a web interface through which users upload the images that are considered as the request to the web instance. The web tier encodes the input image and pushes the binary message to the request queue.

2. **SQS Queues:**
   Amazon Simple Queue Service (SQS) is a fully managed message queuing service that enables you to decouple and scale microservices, distributed systems, and serverless applications.

Request SQS (Standard Queue):

The Request SQS queue holds the binary message of each input image. This binary message is then passed to the App tier for further Image processing.

Response SQS (Standard Queue):

The Response SQS queue holds the processed images output, which is then dequeued by the web tier to display the results to the user.

3. **App Tier:**

   This app tier decodes the binary message from the Request SQS queue and then processes that image using the AMI which holds the deep learning model that predicts the images for input request and then sends the predicted response to the response queue and the S3 output bucket.The maximum number of instances it scales upto is 19.

4. **S3 Bucket:**

   Amazon Simple Storage Service (Amazon S3) is an object storage service offering industry-leading scalability, data availability, security, and performance.

   Input S3 Bucket:

   The Input S3 bucket holds the input images that get processed through the deep learning model.

   Output S3 Bucket:

   The Output S3 bucket holds the response of the predicted image.

2.2 Autoscaling

The maximum number of instances is 19 and one instance is used for running web_tier and controller. The controller constantly monitors the request queue, whenever the queue length is greater than zero first we check if queue length is greater than the number of stopped instances, if it greater then we calculate the number of new instances that have to be created if the stopped instances are less than the maximum number of instances(i.e) 19. If not, we spin up the stopped instances

If the number of stopped instances is greater than the messages in the queue we only spin up the stopped instances equal to the number of messages in the queue. We spawn every instance in a thread so that it can run asynchronously until the image processing completes. We keep track of the threads we initialized.

The instances will keep processing the images until the messages in the queue become empty. As soon as the image processing is done the thread is dead and we check for thread status and if a thread is not alive then the corresponding instance is identified as idle and stopped.

3. Testing and evaluation

Evaluation of output results:

Our code was tested with the workload generator provided by the professor and all the results matched. We were able to generate the output in an acceptable amount of time. The number of objects in the output bucket are matched with the number of objects in the input bucket. The messages in the request queue are increasing to the number of requests sent by the workload generator and then decreasing to zero. The number of messages in the response queue are increasing to the number of requests sent and then decreasing to zero.

Evaluation of auto-scaling:

Auto-scaling was tested against the number of requests that are being sent and our program is able to scale up to 19 instances when the requests are more than 19 and then scale down when the requests are less. It scales up to the number of requests if requests are less than 19 and then scales down when the requests are completed.

4. Code

4.1 process_image.py: This class contains functions to process a certain image by using the model AMI contains. It declares a few variables when we initialize it like the bucket names, queue names, and client objects for global use in the further code. The following are the descriptions of the functions that are present in the code.

4.1.1 fetch_image_from_sqs: Consumes a message from the request queue and deletes it from the queue then the body of the message which contains the image name is returned.

4.1.2 send_image_for_processing: function initiates the processing of an image.

4.1.3 process_image: It fetches a message into the local folder and the path of the image is sent to the face recognition program and invokes the process of uploading the result to the SQS..

4.1.4 download_image: This function downloads the image.

4.1.5 upload_result_to_sqs_response: This function uploads the image results to the response queue.

4.2 web_tier.py: This file contains a class and 2 API endpoints. Flask web server is used. The following are the descriptions of the functions that are present in the code.

4.2.1 The /upload-image route is a POST request definition that uploads an image as a message to the SQS.

4.2.2 The /<file> route is a GET request definition that fetches the object from the SQS and returns it.

4.2.3 upload_message_to_sqs_queue: This function forms a request message and sends it to the request queue.

4.3 controller.py: This file is responsible for actual initiation of the processing of the image files and the autoscaling logic. The following are the descriptions of the functions that are present in the code.

4.3.1 spin_up_instances: This function is an endless loop polling to check if there are any requests to serve. It will be able to spin up 19 EC2 instances maximum depending on the load the request queue is facing. Depending on the state, it will either start new instances or just restart the stopped instances. This function is also responsible for shutting down the idle instances.

4.3.2 execute_each_instance: This function will help the spin_up_instances function to spawn up a process in the EC2's to process the image and upload the result in the output queues (this logic is in the process_image.py file).

4.3.3 get_list_of_running_instances: This function returns the list of running instances.

4.3.4 get_count_of_running_and_stopped_instances: This function will return the number of running and stopped instances.

4.3.5 ec2_instance_config: The EC2 configuration which is required to create new instances is defined.

4.3.6 start_instances: This function will initiate the starting of the EC2 instances with certain parameters and specifications in the code using an AMI given by the professor.

4.3.7 get_list_of_stopped_instances: This function returns the list of stopped instances.

4.3.8 get_queue_length: This function will return the current length of the queue.

4.3.9 process_image_in_ec2: This function is responsible for firing a process in each EC2 to process the image file.

4.4 How to run them ?

4.4.1 The python modules in the requirements.txt file have to be installed.

4.4.2 AWS credentials have to be created in the ~/.aws folder and the access points for queues and buckets have to be given in the code.

4.4.3 Command to start the web-tier:

```
python3 web_tier.py
```

4.4.4 Command to start the controller::

```
python3 controller.py
```

4.4.5 Run the workload generator: NUM is the number of requests you want to upload and PATH is the path to your image input.

```
python multithread_workload_generator_verify_results_updated.py --num_request NUM --url 'http://54.205.77.35:5000/upload-image' --image_folder PATH
```

5. Individual Contribution

Sudarshan Reddy Mallipalli - MSCS - ASU ID: 1222600444

We implemented Face Recognition As A Service on AWS. We used EC2, SQS and S3 services provided by AWS.

Design:

Participated in coming up with the design of the project. Contributed to the design of the processing of multiple images until the request queue is empty once an instance is started. Designed the workflow for the results to be uploaded to the response queue and into the S3 output bucket. Also contributed to how and when the instances have to be killed. Came up with logic on how to take already running instances into account so that we won't scale up more than 19 instances even when other instances are running, and will eventually kill them.

Implementation:

Web-Tier: Implemented uploading the image as a message to the request queue. The image name is read from the file and a message is constructed with a flag string and the image is the filename and encoded image data is pushed into the request queue.

Processing Image: Implemented the function which fetches messages from the request queue and deletes them. As long as the queue delivers messages, the message is sent for processing the image synchronously. When there is no message in the queue the program is stopped to save the resources. As soon as the process is killed the thread which is responsible for running the corresponding instance will be dead, which can be used in the controller to stop the instance. Once the result is obtained from the face_recognition program the output is uploaded into the response queue with the filename and result and pushes one copy to the S3 output bucket.

Controller: Implemented how to track each thread once it is created and to wait until the request queue is empty to start writing the messages from the request queue to the S3 output bucket.

Testing:

First the model was tested with a validator program, to check if the program is returning exact outputs as expected. Tested the auto scaling part with different cases such as having more stopped instances than requests and having instances already running before the start of the controller so as to make sure it won't scale up more than 19.

5. Individual Contribution

Venkata Pavana Kaushik Nandula - MCS - ASU ID: 1223627444

The aim of the project is to develop an elastic application using AWS services which automatically scales-in and scales-out the instances whenever there is a demand. A set of images are sent to the application as the input which predicts the output using the deep learning model and results are displayed. Auto scaling of the instances depends on the load to the application.

Design:

Collaborated with my teammates in designing the workflow of the project. In the web tier, I developed the get and post methods which call the function that uploads the images to the SQS queue. I also worked on the function that starts instances as soon as the input image was received into the SQS queue in the controller.py file. Apart from this, I worked on the process_image.py by writing the functions that process the image.

Implementation:

I did the Implementation of different functions using python, and boto3. Worked on developing the code for get and post methods in the web tier which calls the functions to upload the image to the SQS request queue at the start, and fetching the images that were in the response queue after the prediction is done. Besides this, I contributed to the development of a piece of code that initiates the instances in the controller. I came up with the logic to start a particular number of instances on AWS depending on the messages on the request queue. It connects using boto3 to the ec2 client and then spawns up the EC2's according to a set number of parameters. Took care about specifying the image, security groups and type of instance. I also processed the added process_image.py script that the AMI file which professor gave us. This script consumes the model already present and does additional things like - fetching the messages from the request queue and processing them until the queue is empty.

Testing:

I did a lot of manual testing by running the scripts and checked whether the input data is pushed to the SQS queue using the console. I also thoroughly checked for the start of the instances by varying the input requests and checking whether the instances are created or not with respect to the queue size.