CSE 546 - Project Report

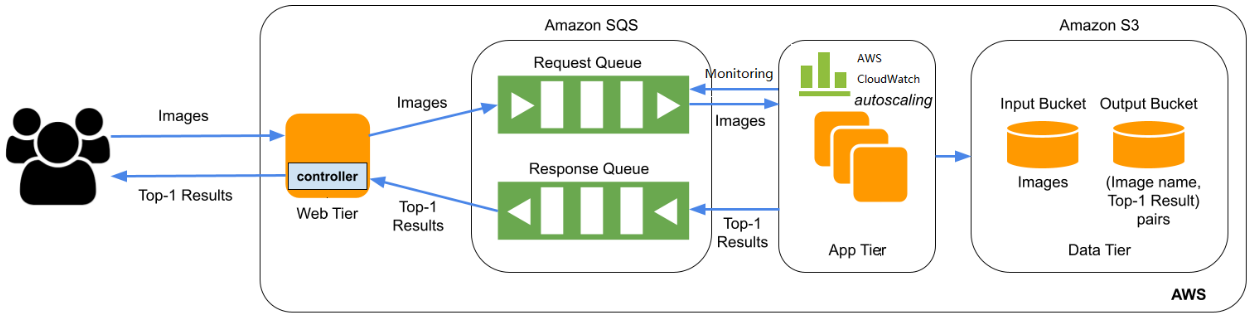
Bhavesh Khubnani, Abhijeet Dixit, Aishwariya Ranjan

1. Problem Statement

This project aims to create an elastic application that dynamically scales out and scales in on-demand, in a cost-effective manner, utilizing Infrastructure as a service (IaaS) cloud technology. Our cloud-based application offers users an image recognition service, employing cloud resources to execute deep learning processes on the images supplied by users to identify them.

2. Design and Implementation

2.1. Architecture



Major components:

Web Tier Controller: The Web Tier allows users to communicate with the application and upload images. The results are also displayed to the users here. The web tier consists of a controller that is responsible for taking user input and receiving output from the Amazon SQS request and response queues respectively.

Amazon SQS: SQS request and response queues are used to pass images as input and classification results as output. They connect the web tier and the app tier.

AWS EC2: EC2 instances are used to run the deep learning model to classify the images. More EC2 instances are launched when demand increases and are terminated when demand decreases.

AWS S3: Two S3 buckets are used, one to store input images from the web tier and the other to store the classification output from the app tier.

2.2. Autoscaling

Explain in detail how your design and implementation support automatic scale-up and -down on demand.

In order to dynamically scale out and scale in our EC2 instances we created a listener that runs as an individual thread to monitor the depth of the SQS request queue every 15 seconds. The depth indicates how many unprocessed messages are in the request queue. Based on the backlog of messages per instance we create more EC2 instances making sure to use no more than 20 instances and making sure that all the pending requests are maintained in the queue once the the app reaches the 20 instance limit. Similarly, if the listener monitors that there are lesser unprocessed images than free EC2 instances, we terminate the instances that are not being used.

2.3. Member Tasks

Bhavesh Khubnani:

Bhavesh was responsible for provisioning the EC2 instance through the AWS console to run our web tier. He designed the web controller for the app which accepted specific image files from the users on the web tier and then uploaded those files to the S3 input bucket and sent the files to the SQS request queue. Further, he coded the scaling logic for our scaling controller where he created a listener to monitor the depth of the queue every 30 seconds and then based on the depth created more or terminated EC2 instances in the app tier to process the images making sure not to exceed 20 instances. He also provided required help to other team members when needed.

Abhijeet Dixit:

Abhijeet provisioned the EC2 instances for the app tier of our application. He also created the web app for users to interact with the application and upload images in the web tier. Using Flask as the backend, he configured the EC2 instances to interact with each other and the internet by setting up the gunicorn (WSGI server) and the NGINX server to proxy requests from the user to the Web Tier. He was also responsible for setting up the SQS queues and creation and configuration of AMI's for the app tier.

Aishwariya Ranjan:

Aishwariya provisioned the request and response SQS queues for the app.Using Python Boto3 provided by AWS, she worked on the code to send messages to SQS request queue and receive messages response queues on the web tier, and on the app tier side to send messages to the response queue and receive them from the request queue, as well as the code to delete messages from the queues. She also wrote functions to create and destroy EC2 instances in ScalingController.py on the web tier.

3. **Testing and Evaluation**

Explain in detail how you tested and evaluated your application.
Discuss in detail the testing and evaluation results

Process: To assess the system we used workload_generator.py. We dispatched various amounts of requests from 1-100 to ascertain the application's capacity to handle the load. Additionally, we consistently monitored the status of the SQS queues to ensure smooth operation. We also monitored the number of EC2 instances being created/destroyed as a result of the change in the depth of the request queue. Finally, we verified the successful storage of images in the designated buckets.

Outcomes: The program executed without any issues, efficiently saving images and their corresponding results in the S3 buckets. Furthermore, instances were dynamically created and terminated based on the SQS queue workload, affirming the correct implementation of our scaling code.

4. **Code**

Explain in detail the functionality of every program included in the submission zip file.
Explain in detail how to install your programs and how to run them.

Web Tier:

AWSController.py

- *upload_to_s3(self, file):* uploads file to input S3 bucket
- *send_to_sqs(self, message):* sends given message to the request queue
- *delete_message(self, message):* deletes the message from the SQS response queue
- *receive_from_sqs(self):* gets a message from response queue and stores the message's content as output_val and returns it.

ScalingController.py

- *check_backlog(self):* checks the depth(unprocessed messages) of the request queue
- *get_instance_map(self)*: Returns the number of running and starting(pending) instances
- *create_ec2_instance(self, cc):* creates a new EC2 instance on being called
- *destroy_ec2_instance(self, destroy_instance_ids)*: terminates the EC2 instance of the id given by 'destroy_instance_ids' input parameter on being called.

- *scale_in_function(self):* determines the EC2 instance to be terminated by calling the get_instance_map(self) function and then terminates it using the destroy_ec2_instance(self, destroy_instance_ids) function.
- *scale_out_function(self, current_instance_count, scale_out_count):* calls the create_ec2_instance(self, cc) function and keeps tract of the current instance count.
- *monitor_queue_status(self):* compares the depth of the request queue and the number of currently running instances and calls the above functions to scale in or scale out accordingly, making sure to not exceed the maximum number of allowed instances.

App.py:

A Flask application used to create the front-end for users to interact with.

- *allowed_file(filename)*: checks if the uploaded file is of the correct type.
- *upload_image_workload()*: It is a POST API endpoint uploads a file of the allowed type to the input S3 bucket and sends it to the request queue and waits for the response. On receiving a response from the response queue it stores it as the output value and returns it, then it calls the function to delete the message from the response queue. This is specifically designed for the multithread_workload_generator.py file Can be accessed via a POST request via http://34.197.236.175/classify route.
- *upload_image()*: This is also a POST API endpoint which is similar to the upload_image_workload() function. This helps the user to view the responses on the web browsers and can be accessed via http://34.197.236.175/ route
- *initiate_scaling()*: calls the monitor_queue_status(self) function from
- ScalingController.py and makes the decision of autoscaling.
- *upload.html*: a HTML page which consists of an option for the users to interact with the app, upload images and view the output

App Tier:

AppController.py

- *get_queue_data()*: gets messages from the request queue and returns it
- *send_data_to_queue(image_output)*: sends the image_output to the response queue
- *download_image(image_name)*: download images from the s3 bucket to the instance's local storage.
- *upload_to_s3(file, filename)*: takes image file as an argument and uploads it to the output S3 bucket.
- *classify_image(image_name)*: takes the image as an argument, runs the classifier code and saves the result