CSE 546 — Project Report

CloudCrowd *(Anvita Lingampalli, Sougata Nayak, Yash Shelar)*

General requirements:

- Strictly follow this template in terms of both contents and formatting.
- Submit it as part of your zip file on Canvas by the deadline.

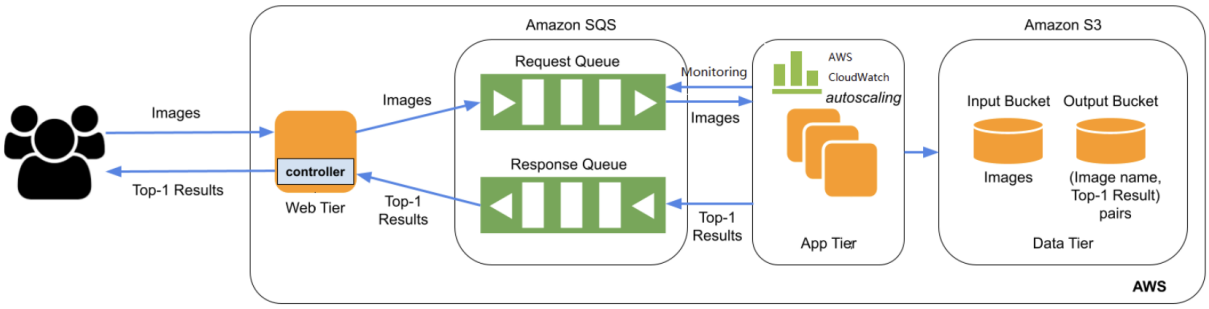
1. Problem statement

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

2. Design and implementation

2.1 Architecture

Reference architecture:



The major components of the architecture are web tier controller, AWS SQS (response and request queues), App tier, AWS CloudWatch, Amazon S3 (input and output buckets)

Services:

AWS EC2: EC2 instances are created using the provided AMI for implementing the webtier and apptier functionalities. More EC2 instances are launched when scaling is necessary.

AWS SQS: SQS request and response queue are used to pass image names as input and classification result as output. They connect the webtier to apptier and viceversa.

AWS S3: S3 buckets are used to store images from the webtier and results from the apptier.

AWS Cloudwatch: Cloudwatch alarms are used to monitor the SQS queue messages to reach a certain threshold.

Description:

Our image recognition app receives images as input and performs image recognition on these images using the deep learning model that is provided. The webtier is responsible for receiving the requests and storing them in S3 input buckets for persistence. The web tier's server also sends the images to AWS SQS. The queues then send the messages to the app tier. AWS CloudWatch is used to monitor the requests in the SQS queue. According to the depth of the queue, the app scales up and down. The deep learning model then performs recognition on the images and stores the images in the output bucket. The SQS request receive the message and sends it back to the web tier and the results are displayed.

2.2 Autoscaling

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

Scaling in and Scaling out is done using simple autoscaling policy with help of cloudwatch alarms. We have created an Autoscaling group named cloudcrowd-asg which contains two policies named policy-1 and policy-2. The policy-1 is used to add an EC2 instance whenever the visible messages in SQS request queue are greater than 1 and the policy-2 is used to remove an EC2 instance whenever the visible messages in SQS request queue are less than 1. For monitoring the SQS request queue, we have set cloudwatch alarms with visibility of messages as the metric. The Autoscaling policy has a desired number of instances as 1 and maximum number of instances set to 19 as requried in the project description. The 1 instance in webtier and 19 instances in apptier make up the 20 total instances stated in the project requirements. The cloudwatch alarms monitor the number of messages every 60 seconds and the autoscaling policies try to launch new instances every 15 seconds as the number of messages scale. Thus, we implemented this simple autoscaling policy using cloudwatch alarms.

2.3 Member Tasks

Explain in detail what is each member's task is in the project.

Anvita Lingampalli:

Design: According to the architecture and design of the application, I was responsible for implementing the code behind the web tier. For convenience and efficiency, we are uploading the images into S3 bucket from the web tier instead of app tier.

Implementation: I implemented the code which was responsible for receiving the http requests from the workload generator that was provided. The image files were then sent to s3 input

bucket. I wrote the functionality for storing the images in the input bucket with the help of AWS documentation. The web tier also calls the function which sends the message to the request queue. It also receives the message from the response queue and displays it.

Testing: I was testing and making sure that the web tier was fetching the requests correctly, and the images were uploaded to the s3 bucket. I also tested if the web tier was receiving and sending back the responses correctly.

Sougata Nayak:

Design: I was responsible for designing and implementing the App tier code. The app tier code involved fetching the images from the SQS queues, running the classification algorithm on those images, storing their results in an S3 bucket and sending back those images to the response queue.

Implementation: Using the Python Boto3 provided by AWS, we implemented the app-tier code. When creating the Auto-Scaling groups, we had to add User Data commands to install boto3 so that we are able to use them in out program. Boto3 allowed us to access aws functionalities using python code. Using boto3, we were able to access the sqs queue we used for input and fetched the images that needed to be classified. For the classification of images, a python script named image_classification was provided in the AMI which was used. Then I stored the result of the classification in a file which was named image_name.txt. Then I would read the result of the classification and store it in the S3 output bucket using the boto3 functions. Finally I would send the result with the image name to the response queue in sqs so that the web tier can get the results. In my main code, I created a while loop which runs indefinitely until stopped, so that after each iteration, we would fetch a new image and do the entire process again. The while loop will only be broken if the number of items in the sqs queue becomes zero.

Testing: For testing, we checked the the classifier is working properly even when multiple images are being sent by the sqs request queue. Also checked the time it takes for the completion of each classification.

Yash Shelar:

Design:

The design required two distinct SQS queues and they were the main bridge between both ends of the project. I was responsible for the same and designed them based on examples found in the AWS documentation.

Implementation:

My job was to implement the SQS request and response queues. I used the documentation provided by Boto3 and AWS to implement the necessary functionality. Main functions created where to send and receive requests and responses from the webtier to apptier and vice versa. I was also responsible for creating the autoscaling policy and set up cloudwatch alarms which would monitor the SQS queue for visible messages and add instance or delete instance based on a certain threshold.

Testing:

I helped in monitoring the cloudwatch alarms and the EC2 instance scaling while testing. I was making sure there were messages in the SQS queue and the cloudwatch alarms were getting triggered at certain thresholds. I was also managing the security groups so that the scaling policy an launch configuration of the EC2 instances are able to be handled from our own devices.

3. Testing and evaluation

Explain in detail how you tested and evaluated your application.

Discuss in detail the testing and evaluation results.

Testing and Evaluation steps and strategies: For testing we used both the workload_generator.py and the multithreaded_workload_generator.py and sent requests in various ranges from 1 to 100 to make sure the application was able to handle the load. Also we kept checking the status of the SQS queues regularly to monitor the flow was working correctly. We also regularly kept a check on the number of instances created by our Auto-Scaling group and checked if it is based on the the Cloudwatch alarms we set for them. Finally we verified that the images are successfully being stored in the buckets.

Testing and Evaluation results: The program ran successfully and stored the images and the result in s3 buckets. Also the instances were being created and terminated depending on the number of items in the SQS queues which meant the Cloudwatch monitor was working correctly.

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.

WebTier:

Index.js:

Index.js file receives the request from the workload generator and calls the function to store image in S3 bucket and calls the function which sends message to SQS request queue. It also receives the messages from the response queue and sends the response.

storeImageinS3.js:

storeImageinS3 file has the function which receives the images and stores it in the S3 input bucket (cloudcrowd-input bucket).

sqsWebtier.js

- getQueueURL(): it takes the required queue name as string and returns the queueURL
- getQueueData(): it takes the queue name and returns the approximate number of messages in the queue and last modified timestamp
- sendMessage(): it takes the message (image name) to be sent to request queue and the queueURL and returns aws queue request
- receiveMessage(): it takes the response queue url and returns the messages in the queue
- deleteMessage(): it takes the response queue url and deletes the messages from the queue

AppTier:

App-tier.js

- get_sqs_url(): It takes the queue name and returns the sqs queue's url
- read_message(): it takes the sqs queue's url as an argument and return the messages from that queue
- download_images_from_s3(): download images from the s3 bucket to the instance's local storage.
- classify_images(): The actual function which takes the image as a argument, runs the classifier code and saves the result.
- write_message_to_response(): This function is responsible for sending the result of the classification to the response queue.
- send_classification_result_to_response_queue(): This function reads the output of the classification from the output file and sends it to the write_message_to_response() function to send to sqs.
- write_response_to_bucket(): This function writes the output of the classification to the output bucket.
- delete_image(): Delete the image from local storage.
- delete_message_from_resuest_queue(): This function deletes the message from the request queue so that other instances don't work on the same image.