CSE 546 — Project 1 Report

Student Names: Raviteja Reddy Guntaka, Sai Manoja Gadde, Karthik Chadalavada

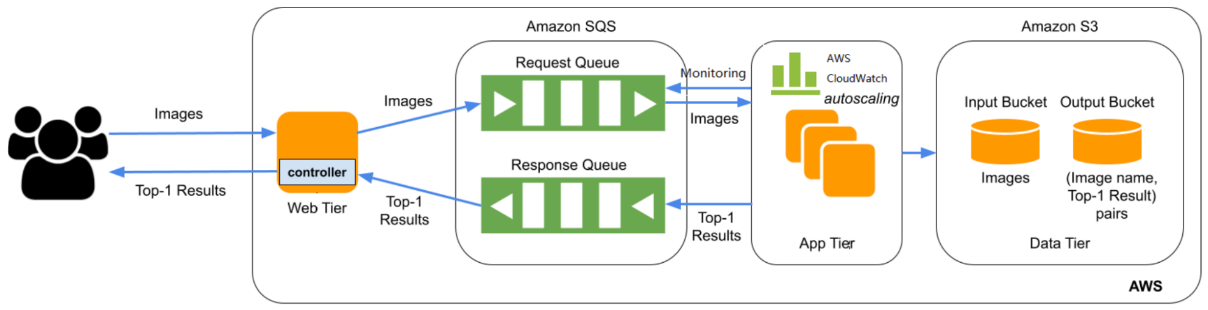
1. Problem statement

With the increase of digital images and the need for automated image processing and recognition, there's a growing demand for efficient and scalable image recognition systems. These systems often need to handle large volumes of image data, requiring scalable infrastructure that can adapt to varying workloads.

Develop an elastic cloud-based image recognition application utilizing a provided deep learning model. This application should harness the IaaS capabilities of Amazon Web Services (AWS) to ensure automatic scalability in response to varying demand. It must efficiently handle concurrent requests, scale in accordance with the depth of the Request SQS Queue as monitored by auto scaling logic, and consistently store all input and output data in S3. The system's effectiveness and efficiency will be measured using a workload generator, with an imperative to process all requests promptly and accurately without deploying more than 20 instances.

2. Design and implementation

2.1 Architecture



a. **WebTier:** The web tier is crucial for our cloud-based image recognition application as it serves as the primary interaction point for users and facilitates the flow of data through the system. It is divided into two main components: the Input Receiver and the Output Generator.

Input Receiver - Python Flask API:

- **Function:** The Input Receiver is responsible for accepting image uploads from users and initiating the processing workflow.
- **Endpoint:** It exposes an "upload" endpoint where users can send a POST request with their image.
- **S3 Integration:** Upon receiving an image via the POST request, the Flask application immediately stores this image in an AWS S3 bucket. Using S3 ensures that the data is stored securely and durably, allowing for easy retrieval for processing.
- **Request Queue (SQS):** After the image is safely stored in S3, a message referencing the image location (S3 URL) is placed into an Input Queue. This queue serves as a buffer, holding requests until they can be processed by the application instances.

Monitoring the Queue: An auto-scaling script continuously monitors the depth (number of messages) of the Request Queue (SQS).

- Based on pre-defined logic (e.g., if the number of messages exceeds a certain threshold), the script makes decisions about scaling the processing resources. This ensures that as demand increases, the system automatically scales out (adds more instances) to process the incoming images promptly.
- Conversely, during low demand periods, the system can scale in (terminate extra instances) to conserve resources and minimize costs.
- The auto-scaling ensures that the number of instances does not exceed the specified limit (in this case, 20 instances).

Output Generator:

- **Function:** The Output Generator is tasked with retrieving processed results from the system and conveying them back to the user.
- **Response Queue (SQS):** After the processing tier concludes its image recognition task, the results are placed in a Response Queue.
- **User Communication:** The web tier continuously monitors this Response Queue. When a relevant processed result is detected, the application fetches the message, deciphers the data, and then instantly relays the outcome to the user through the same "upload" endpoint, ensuring a synchronous user experience.

b. **App Tier:** The App Tier, also known as the processing or application tier, is pivotal in the execution of the core functionalities of our cloud-based image recognition system. It's primarily responsible for processing user-submitted images and producing classification labels through the provided deep learning model.

Initialization and AMI Boot-Up:

- **Function:** Whenever a new instance is spun up using the Amazon Machine Image (AMI), it initiates the core processing environment for the application.
- **Python Script Execution:** As part of its initialization, the instance automatically runs a Python script. This script sets up the necessary environment and dependencies, ensuring the instance is ready for image processing.

Listening to Request Queue:

- **Function:** The primary task of the App Tier is to continuously monitor the Request Queue (SQS) for incoming image processing requests.
- **Message Processing:** Once a message (indicating a new image for processing) is detected in the queue, the instance fetches the message. The message typically contains a reference to the image location stored in the S3 bucket.

Image Processing and Classification:

- **Function:** After retrieving the image based on the reference from the message, the instance processes it using the provided deep learning model. Classification Label Generation: The model analyzes the image and determines a classification label, representing what the model recognizes or interprets from the image.

Output Generation and Response Queue:

- **Output File:** Once the classification is complete, an output file is generated. This file contains details of the processed image and its corresponding classification label.
- **Response Queue Integration:** The details from the output file, or a reference to it, are then placed into the Response Queue (SQS). This queue serves as a bridge between the App Tier and the Web Tier, notifying the latter of the completed processing task.

In summary, the App Tier stands as the operational heart of the system. It seamlessly integrates with both the Web Tier and data storage solutions, ensuring images are processed efficiently and results are relayed back in a timely manner. The use of SQS ensures that the system can handle varying loads, allowing for scalability and ensuring responsiveness regardless of the volume of incoming requests.

2.2 Autoscaling

Auto-scaling is a vital component of our cloud-based image recognition system, ensuring that the App Tier dynamically adjusts its resources based on the incoming demand. This ensures both cost-effectiveness and high responsiveness. Let's delve deeper into how the auto-scaling is designed and implemented:

Setting Boundaries:

- **Minimum Instances:** The system is configured to always have at least 1 instance running, even during periods of inactivity or low demand. This ensures that the system remains responsive and can immediately start processing when new requests come in.
- **Maximum Instances:** To control costs and manage resources, a hard cap is set at 20 instances. This ensures that while the system can scale out to handle bursts of demand, it doesn't do so infinitely and lead to unforeseen expenses.

Tracking Request Queue:

- **Function:** Central to the auto-scaling mechanism is the continuous monitoring of the Request Queue (SQS). The depth (or message count) of this queue provides a real-time indicator of the system's current workload.
- **Scaling Trigger:** As messages (representing image processing requests) accumulate in the queue, they act as triggers for the auto-scaling action.

Load Factor and Scaling Decision:

- **Defining Load Factor:** The load factor is a predefined constant which, in our system, is set to 5. This essentially means that each EC2 instance in the App Tier is expected to handle 5 messages or image processing tasks concurrently.
- **Scaling Logic:** The system calculates the required number of instances by dividing the current message count in the Request Queue by the load factor. For example, if there are 25 messages in the queue, the system deduces that it needs 5 instances to handle this load (25/ 5 = 5).
- **Scale-Up:** If the computed required number of instances is greater than the currently active instances and doesn't exceed the maximum limit, the system scales out by launching additional instances from the App Tier AMI.
- **Scale-Down:** Conversely, if the system detects that the message count has decreased and some instances are underutilized (handling fewer than 5 messages), it will terminate the excess instances, ensuring resources are conserved and costs are minimized. However, it will always maintain at least the minimum number of instances.

In essence, the auto-scaling mechanism in our design is both dynamic and efficient. By continuously monitoring the Request Queue and using the load factor as a guide, it ensures the system is always equipped with just the right amount of resources to handle the incoming demand, balancing both performance and cost.

2.3 Member Tasks

Sai Manoja

- **Elastic IP Creation:** Responsible for setting up the Elastic IP, ensuring that the system has a static IP address. This is crucial for maintaining a consistent endpoint, especially for systems that need to scale and might have instances being replaced.
- **Web Tier Development (web_tier.py): API & Flask App Setup:** Took the lead in creating the primary user interface of the system. Using Flask, designed and implemented the API that users interact with to upload images and receive results.
- **S3 Integration:** The logic to securely store user-uploaded images in AWS S3 buckets was developed. This ensures that images are stored durably and can be retrieved efficiently for processing.

Raviteja Reddy

- **Auto-scaling Implementation:** Focused on the system's dynamic scaling capabilities. Set up the rules, alarms, and policies to ensure that the system can scale its resources in and out based on demand. SQS **Integration in Web Tier:**
- **SQS Publisher:** Implemented the logic in the web tier to push messages to the Request Queue (SQS) whenever a new image is uploaded. This ensures that the processing tier is aware of pending tasks.
- **Testing:** Once the S3 integration was complete, conducted tests to verify the correct storage and retrieval of images.

Karthik

- **App Tier Development (app_tier):** Was responsible for the core processing environment of the application. He set up the instances to run the Python scripts, retrieve images from S3, process them using the deep learning model, and then communicate the results back via the Response Queue.
- Ensured that the App Tier instances could efficiently communicate with both the Web Tier and data storage solutions, ensuring images are processed efficiently and results are relayed back in a timely manner.

By dividing tasks in this manner, each member played a crucial role in ensuring the system's functionality, scalability, and efficiency. Their combined efforts resulted in a robust and dynamic image recognition system.

3. Testing and evaluation

We started testing the initial setup of the flask app which is hosting the endpoint used. We resolved multiple issues related to accessing the flask app endpoint running on our web-tier instance. Created an Elastic IP to overcome this problem. Later, we focused on testing web-tier and app-tier together. Used workload_generator code to hit image requests. As this is a sequential process(workload generator hits one request at a time), auto scaling isn't needed much. The minimum instances are sufficient to satisfy the load.

This resulted in fine tuning our web-tier and app-tier applications further. After finishing initial testing, we started using multithreaded_workload_generator to look at the auto scaling behavior.

Started with 100 requests, the application needed around 7-8 minutes to finish processing all the requests and the variation in message count, running instances are monitored closely. It is observed that at peak load the app requested around 15-20 instances to complete the image classification task.

The outputs are monitored following logs on the particular instances. And the logs suggested consistent behavior.

4. Code

web-tier.py - This is a multi threaded Python flask application.

- It is exposing a '/upload' POST method which can be hit by a user.
- It also holds the logic for continuous polling of the response queue.
- This continuous polling helps in responding to the messages received on the response queue.

app-tier.py - This is a python which continuously polls the request queue.

- This continuous polling helps in responding to the messages received on the request queue.
- Once a message is received, the corresponding image is downloaded from S3 and further processed through the classification algorithm.
- Once classification is done, the result is written to a file and uploaded to S3.
- Once the upload is successful, a message is published to response queue and the current message is deleted from request queue.

auto_scaling.py - This file holds the logic for auto scaling.

- The app-tier application is saved to an ami and a shell script (startup_script.sh) is written to start the app-tier.py once an instance is launched.
- This initially reads the number of messages on the request queue and the running ec2 instances.
- A ratio of 5:1 is taken as the base to spin up instances. Meaning if there are 25 messages to be processed on the queue, the script calculates a need for a min of 5 instances to meet this load.
- As per the calculation it decides whether to SCALE OUT or SCALE IN the number of instances. And accordingly launches/terminates the said number of instances.
- This script makes use of the AMI id in which the app-tier.py, startup_script.sh and the crontab are configured.

Extract the contents of the zip folder to a directory and open a command line/terminal in that directory.

- Use this url as the api endpoint when running the workload generator.
  http://34.194.131.161:5000/upload
  python .\multithread_workload_generator.py --num_request 100 --url http://34.194.131.161:5000/upload --image_folder .\SampleImages\
- The AMI used is - ami-0eb335b25f6d10761
- To verify the crontab open any instance which is created for app-tier and type
  crontab -e
- **Web-tire.py is run using /usr/bin/python3 /home/ubuntu/web-tier/web_tire.py**