CSE 546 — Project - 1 Report

Sai Varun Reddy, Mullangi
Siva Nagi Reddy, Munaganuru
Pavan Krishna Reddy, Madireddy

1. Problem statement

The problem being solved in this project is providing image recognition as a cloud service using a deep learning model provided by the client. The service will take in images in the PNG format from users, use the deep learning model to perform image recognition, and return the top prediction from the model to the user as plain text. The importance of this service is to provide a scalable and efficient solution for image recognition that can be accessed remotely by users, without the need for them to train their own models or have access to specialized hardware. This can be useful for a variety of applications, such as identifying objects in images for e-commerce, detecting anomalies in medical images, or analyzing satellite imagery for environmental monitoring. The service also aims to handle multiple requests concurrently and automatically scale up or down based on demand, ensuring efficient resource utilization while maintaining the performance and availability of the service.

2. Design and implementation

2.1 Architecture

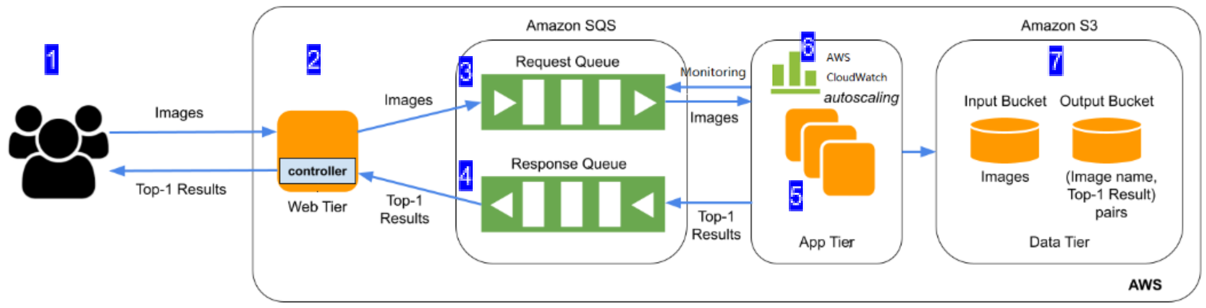

Fig 1- Architecture of the Image Recognition Service

Above picture is the overall higher level with all major components of our implementation of this project. Each component functionalities are as follows.

Component 1 - Workload generator:

To test the speed and accuracy of the image recognition cloud service, we use a concurrent workload generator that enables us to make multiple requests simultaneously.

Component 2 - Web Tier:

The web tier is the component of our application that is exposed to users. In this tier, we use a Flask server to handle HTTP requests that include image files. The web tier performs two critical tasks.

First, it encodes the image and sends it to the SQS Request Queue (component 3), keeping the connection open until a message is received from the SQS Response Queue (component 4). To avoid the difficulty of matching fetched messages from the response queue with multiple threads created by Flask, we have created a background thread dedicated to polling the SQS Response Queue (component 4).

Second, another background thread constantly polls the SQS Response Queue (component 4) for messages and stores them in a local dictionary with a unique key based on the image name. User threads can then read messages from this dictionary and send them back to the user.

By separating these tasks into dedicated threads, we ensure that our web tier can efficiently handle incoming user requests while still effectively managing the SQS Response Queue (component 4).

Component 3 - SQS Request Queue:

We use Amazon Simple Queue Service (SQS) to receive messages from the Web Tier. The messages are then polled in the App Tier (component 4) and deleted.

Component 4 - SQS Response Queue:

We use Amazon Simple Queue Service (SQS) to receive messages from the App Tier. These messages are polled in the Web Tier (component 2) and deleted once a match is found with the corresponding message sent to the Request Queue.

Component 5 - App Tier:

We have created an image that contains the pretrained deep learning model. Each instance is booted with this image, and the Python script triggers the loading of the model in "eval" mode to avoid repeated loading. The App Tier continuously polls the messages from the Request Queue (component 3). Upon receiving an image to classify, the message is deleted from the Queue, and the model is used to classify the image. Once the classification is complete, the image is stored in the input S3 bucket, and the classification result is stored in the output S3 bucket (component 7).

Component 6 - AutoScaling:

We have implemented AutoScaling to automatically adjust the number of instances running in response to changes in traffic to ensure optimal performance. In this architecture, the change in traffic can be estimated by number of messages in the Request Queue available. This is the best metric available to scale the app tier as these messages are the backlog which must be processed and cleared from the queue. We have created an autoscaling process which gets a custom metric which is **ApproximateNumberOfMessagesVisible / GroupInServiceInstances** from AWS Cloudwatch using boto3 client. The custom metric gives insight of the number of messages in Request queue which will eventually be processed by the number of available app tier instance. Based on the data extracted from cloudwatch, the autoscaling application updates the EC2 autoscaling group’s desired state which indicates the number of instances of app tier needs to be up and running. Once the desired state is updated in the autoscaling group there will be either creation of new instance or termination of running instances to match the desired state.

Component 7 - Amazon S3:

We use S3 buckets to store the information from the App Tier (component 5), and we access the buckets from the App Tier using the botoclient.

2.2 Autoscaling

The design of autoscaling is based on the cloud watch metrics. The main idea of autoscaling is to improve the performance and reliability of the web tier and make sure that the EC2 instances are scaled according to the needs. Keeping this in mind, we are using ASGs (Amazon Autoscaling groups) which keeps track of the EC2 instances by maintaining instance count value equal to the desired count.

The Autoscaling group for the app tier is created and configured in such a way that it has minimum of 1 instance unless there is a manual override of the desired count to 0. The maximum is set to 20 meaning that it can spin up 20 EC2 instances. The main part of autoscaling is performed by a python process which keeps polling a custom metric from CloudWatch, which is **ApproximateNumberOfMessagesVisible / GroupInServiceInstances** using a boto3 client. The custom metric gives insight of the number of messages in Request queue which will eventually be processed by the number of available app tier instance. This is the perfect metric which indicates the load on each instance and the need to upscale/ downscale at any point of time. There are proper thresholds defined crossing which the script triggers an update to the desired capacity in the Autoscaling group of the app tier which accordingly increases or decreases the number of instances running. One of the notable additional features set in the Autoscaling group is the Cooldown period, which signifies the period after which the next scaling can happen. This has a notable impact on the performance of autoscaling as this period gives enough time for the scaling process to complete and then reflect the same in the cloudwatch metrics.

2.3 Member Tasks

Munaganuru Siva Nagi Reddy - Component (2,3,4)

As part of my project responsibilities, I took charge of configuring the web instance by installing all necessary dependencies, including Flask. I was also responsible for developing the core functionality of the web tier, which involved receiving HTTP requests from users with "myfile" as the key for the image. The web tier then processed the image and sent it to the SQS. I ensured that all interactions between the web tier and SQS were seamless and concurrent threads were handled effectively. Additionally, I implemented a background process to continuously poll the response queue and store response messages in a dictionary. This approach ensured that user threads were not blocked while waiting for responses. Furthermore, I leveraged system services and created a new service to perform a Git pull and keep the web tier up-to-date with the latest changes.

Sai Varun Reddy Mullangi - Component (5, 7)

I had several crucial tasks to perform. Firstly, I was responsible for preparing the image for classification. To do this, I decoded the image data that was received from the request queue and used the Python Imaging Library (PIL) to open the image. Then, I transformed the image so that it could be used as input to the pretrained deep learning model. Additionally, I implemented the function to store the image in the input S3 bucket. This ensured that the image was properly stored and could be accessed later if needed. Secondly, my role was to classify the image using the pretrained deep learning model. To do this, I loaded the model and used it to classify the transformed image. I also implemented the function to store the classification result in the output S3 bucket.

Lastly, I was responsible for monitoring the process of image classification. I implemented a signal handler function to handle the SIGTERM signal and perform any necessary cleanup operations. This was important to ensure that the process of classification was performed efficiently and with minimal errors.

Additionally, I added logging statements to monitor the process and to log when an image had been predicted. This provided useful information to users and helped in debugging any issues that may have occurred during the classification process.

Pavan Krishna Reddy Madireddy – Component (1, 6)

For implementing the Autoscaling solution for an application that is hosted on Amazon Web Services (AWS). It had two main requirements on a very high level:

To ensure that the number of running instances is always in line with the number of incoming messages to an SQS queue and to ensure that the scaling operation does not take more than 7 minutes. To meet these requirements I have tried two approaches.

Approach 1: Using CloudWatch Alarms

The first approach involved setting up CloudWatch Alarms to monitor two custom metrics. The first metric monitored the number of messages in the SQS queue. The second metric monitored the number of instances currently running in the Auto Scaling Group (ASG). If the number of messages in the queue exceeded a certain threshold, the CloudWatch Alarm triggered an Upscaling policy to add more instances to the ASG. Similarly, if the number of messages in the queue decreased below a certain threshold, the CloudWatch Alarm triggered a Downscaling policy to remove instances from the ASG.

While this approach worked, the problem was that downscaling took 15 minutes and upscaling took 5 minutes. This was because of the "period" attribute in the scaling policy, which was not configurable. Since this was taking more than the threshold of 7 minutes indicated in the project requirements, an alternative approach was tried.

Approach 2: Using a Python Script

Approach 2 is a Python script that runs continuously and monitors the SQS queue's approximate number of visible messages and the number of instances running in the Auto Scaling Group (ASG) using the CloudWatch metric data. The script compares these metrics to predefined thresholds and scales up or down the ASG based on the comparison result.

The script starts by importing the necessary libraries and setting up the AWS SDK clients for CloudWatch, Auto Scaling, and EC2. It then enters a continuous loop that periodically retrieves the CloudWatch metric data for the defined period of the last 5 minutes, with a 30-second granularity. The metric data is retrieved using a CloudWatch GetMetricData API call, which allows for querying multiple metrics at once and performing calculations on the retrieved data. The script then calculates the average number of messages per instance to determine if the ASG needs to be scaled up or down. The calculated value is then compared to the predefined thresholds to determine the desired number of instances in the ASG. These thresholds can be adjusted to suit the specific needs of the application. The script then retrieves the current desired capacity of the ASG and If the desired capacity is not equal to the calculated desired capacity, the script updates the desired capacity of the ASG using the Auto Scaling. After updating the desired capacity, the script sleeps for 30 seconds before starting the loop again to ensure that the new instances have time to spin up before checking the metrics again.

Overall, the second approach provides a more flexible and customizable solution for scaling the ASG based on the SQS queue's message count, allowing for fine-tuning of the scaling thresholds and reducing the scaling time to a few minutes, as opposed to the 15-minute minimum for CloudWatch alarms.

3. Testing and evaluation

Based on the load testing results, it can be concluded that the system was able to handle a considerable amount of load without any major issues. With 100 requests sent to the URL, the Flask server was able to send all the messages almost concurrently to the SQS queue. The scaling process was able to detect the increased load and scale out to a maximum of 19 instances in under 2 minutes, allowing the system to handle the increased load effectively. The client received all 100 responses in 4 minutes, indicating that the responses were being processed efficiently. Finally, the scaling process scaled in to 1 instance in 6 minutes, indicating that the system was able to handle scaling in as well. Overall, the system performed well during load testing, and the scaling process was able to handle the load effectively.

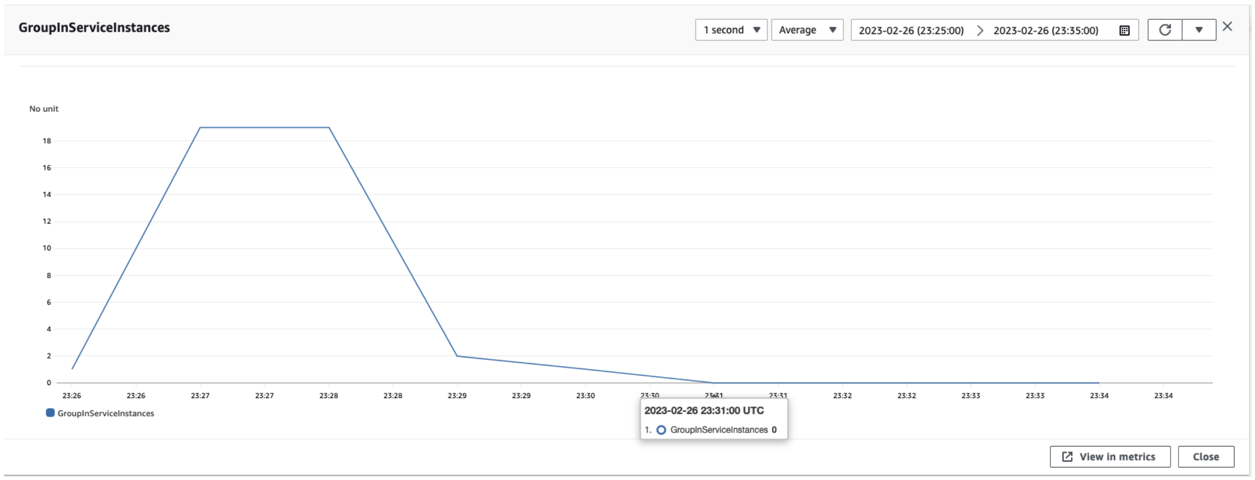

Figure 2 shows the plot between Number of Instances vs Time

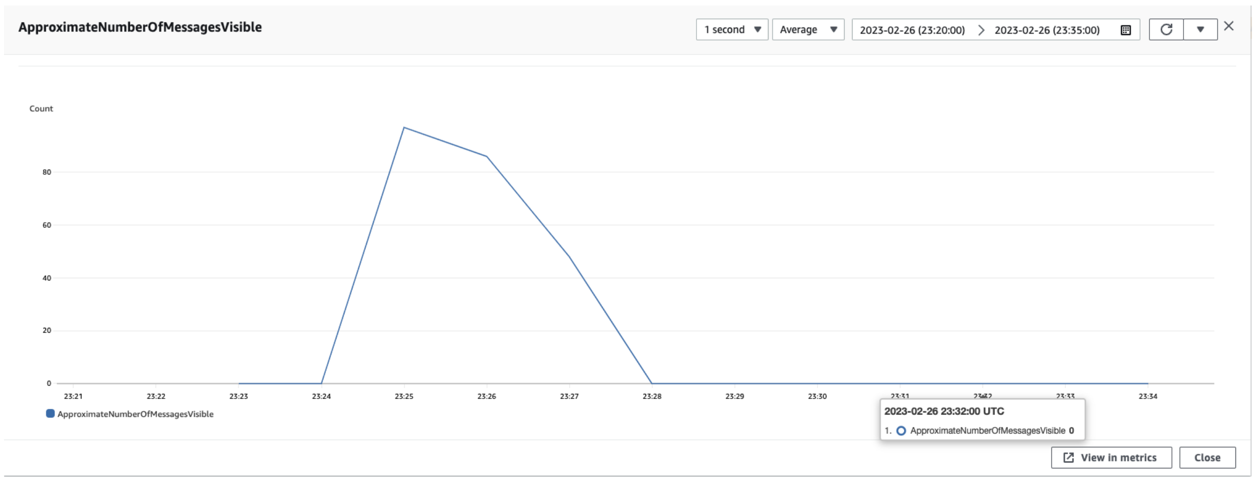

Figure 3 shows the plot between Number of messages visible vs Time

4. Code

Web Tier:

We have a python script *app.py* in this component. Functionality of this script is as follows.

It is a Python Flask web application that exposes an API endpoint **/classify** for image classification. The application uses an Amazon SQS queue to send requests for image classification and receive responses.
When a client sends an image to the **/classify** endpoint, the image is first sent to the request queue and a background thread is started to continuously poll the response queue for a response message corresponding to the client's image.
The main thread waits for the response event to be set by the background thread, indicating that a response has been received for the client's image. Once the response is received, it is returned to the client.

App Tier:

We have a Python script that listens to an Amazon SQS (Simple Queue Service) request queue, receives a message containing an image, applies image classification to the image, and then stores the classification result in an Amazon S3 (Simple Storage Service) bucket.

The script first imports necessary Python libraries including Flask, PyTorch, PIL (Python Imaging Library), Boto3 (an Amazon Web Services (AWS) SDK for Python), and others.

Next, it defines two functions that store an image and its classification result in an Amazon S3 bucket, respectively. These functions use the Boto3 library to interact with the S3 bucket.

After defining these helper functions, the main function classify_from_queue loads a pre-trained ResNet18 model from PyTorch and uses it to classify the received image. This function first transforms the image into a tensor and passes it through the ResNet18 model to obtain the classification result. Then, it loads the labels from a JSON file and maps the predicted label index to its corresponding label text. Finally, it returns the classification result as a string.

The script then defines a signal handler that can catch the SIGTERM signal (termination signal) and perform any necessary cleanup operations.

In the main block, the script continuously listens to the SQS request queue and waits for incoming messages. Once a message is received, it extracts the image data and its name from the message body and stores the image in the input bucket. Then, it calls the classify_from_queue function to classify the image and stores the classification result in the output bucket. The script then sends a message containing the image name and its classification result to the response queue and deletes the original message from the request queue.

Overall, this script serves as an image classification service that can be deployed in a cloud environment such as AWS, where incoming images can be processed and their classification results can be stored in an S3 bucket.

AutoScaling:

This code is designed to monitor the number of messages in an SQS (Simple Queue Service) queue, and the average number of instances in an Autoscaling group over a 5-minute period. It then calculates the backlog per instance by dividing the number of visible messages in the queue by the average number of instances in the Autoscaling group.

We are using the Boto3 Python library to interact with the AWS (Amazon Web Services). Specifically, it uses the CloudWatch, Autoscaling, and EC2 client objects to retrieve metrics and perform scaling operations.

The loop in the code runs continuously, with a 10-second pause between each iteration. Within each iteration of the loop, the code uses the CloudWatch client object to retrieve the number of visible messages in the SQS queue and the average number of instances in the Autoscaling group over the past 5 minutes. It then calculates the backlog per instance using an expression that divides the number of visible messages in the queue by the average number of instances.

Based on the backlog per instance, it determines the desired number of instances in the Autoscaling group. If the desired number of instances differs from the current number of instances in the group, the Autoscaling client object is used to update the desired capacity of the group accordingly.