CSE 546 — Project Report

Image Classification over Cloud

Sai Vikhyath Kudhroli - 1225432689

Gautham Maraswami - 1225222063

Abhilash Subhash Sanap - 1225222362

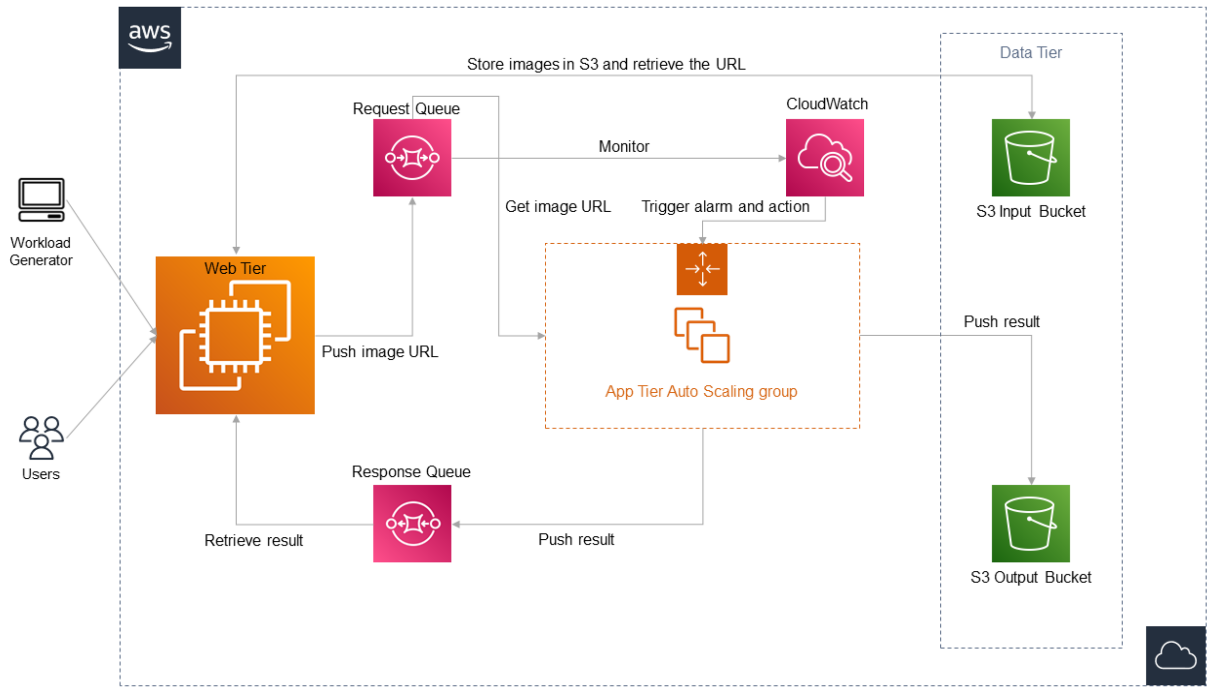
1. Problem Statement

The main objective of this project is to develop an elastic application that can scale in and scale out based on the demand to perform image classification on AWS IaaS resources. A deep learning model that performs image classification is deployed on the AWS IaaS resources so that varying demands from the users can be handled effectively without incurring much cost. A traditional application is inefficient to handle varying demands and is costly to add new resources to handle varying demands. Hence, a cloud-based solution is vital for providing reliable and high-availability service to users which we aim to develop in this project.

2. Design and implementation

2.1 Architecture

The system comprises five major components: a web tier, a messaging layer, an app tier, a cloud watch, and a data tier. All these components are collectively responsible for an elastic application that provides an image recognition service. Each of the components is explained in further detail below:



2.1.1 Web Tier

The web tier receives multiple requests from users or the workload generator and sends the appropriate response back. The web tier consists of a single AWS EC2 instance. A REST API is built in the web tier which receives the request from various users, or the workload generator extracts the image from the request and pushes this image to input S3 bucket. Once, the image is pushed into the input S3 bucket, the web tier retrieves the URL with which the image can be accessed. Then this URL along with a unique key to identify the session is pushed into the request queue. When the results are available in the response queue, the web tier pulls them from the response queue, maps the result to the appropriate session based on the key, and sends them back to the appropriate user or the workload generator. The REST API running on the web tier is built using Flask which continuously listens to the requests. Boto3 AWS SDK is used to push the image to the input bucket, fetch the URL, push the URL of the image to the request queue, and retrieve the response from the response queue. It is also responsible for deleting the response from the response queue after reading it. Uninterruptible execution of the REST API is supported by the no hang up (nohup) process which is scheduled as a cron job that runs every time the EC2 instance is rebooted. This ensures the high availability of the REST API, meaning the web tier is down only when the EC2 instance is down.

2.1.2 Messaging Layer

The message layer consists of two SQS queues, one named request queue and the other response queue. The request queue is an autoscaling and reliable messaging queue service that stores the image URL along with the unique key pushed by the web tier until it is picked up by the app tier. The URLs are stored in the request queue until they are read and deleted by the app tier. The response queue is also an autoscaling and reliable messaging queue service that stores the classification result along with the unique key pushed by the app tier until it is picked up by the web tier. This response is stored in the response queue until they are read and deleted by the web tier. Both queues are responsible for maintaining the synchronization between the web tier and the app tier.

2.1.3 App Tier

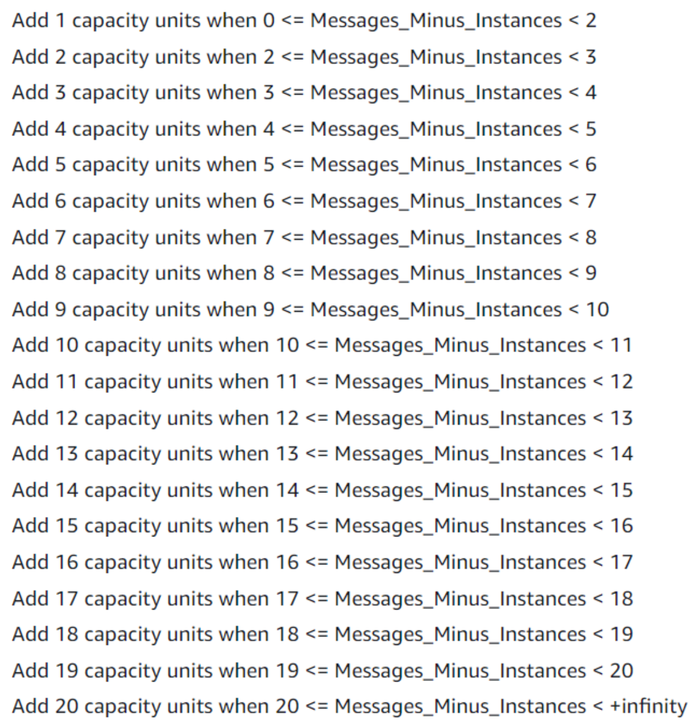
The app tier is responsible for fetching the image URLs and the unique keys pushed by the web tier from the request queue, triggering the classification function with the image URL as an argument, and obtaining the classification result. This classification result is pushed into the response queue along with the unique key obtained from the request queue. The classification result obtained is also pushed into the output S3 bucket in the form of a key-value pair of the image name and the classification result. The app tier continuously tries to fetch messages from the request queue when running. A launch template is created using the image of the instance which consists of the app tier code. Boto3 AWS SDK is used for fetching the messages from the request queue, pushing the results into the response queue and the output S3 bucket. This launch template is used to spawn instances when autoscaling is triggered. The automatic start of the app tier script is triggered by the cron job which gets automatically triggered when an instance is booted. Thus, responsible for starting the app tier script every time a new instance is spawned.

2.1.4 Cloud Watch

AWS cloud watch is used to monitor the number of messages and the app tier instances in the auto-scaling group, which are collectively used to trigger the autoscaling of instances in the app tier. Cloud watch consists of two alarms that monitor the number of messages minus the number of instances which is used as the metric to trigger scale-up or scale-down action. This is explained in detail in the Autoscaling section.

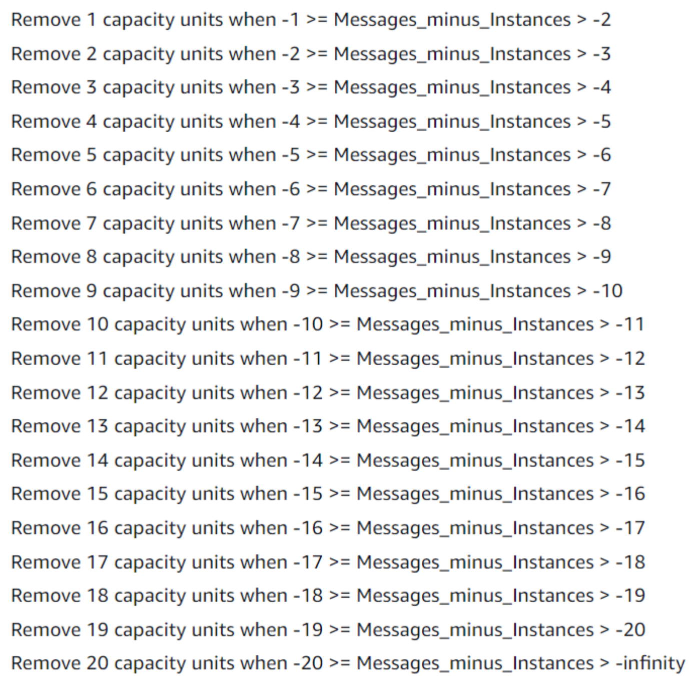
2.2 Autoscaling

Policy - Step Scaling using a custom metric. It is aimed at adding as many EC2 instances as there are messages in the queue and removing the EC2 instances that are in excess when compared to messages in the queue. The minimum capacity is set to 1 while the maximum capacity is set to 20.
Custom metric - ApproximateNumberOfMessagesVisible (q) - GroupInServiceInstances (i)
Scale-Out
CloudWatch Alarm condition: q - I > 0
Actions:



Add 1 capacity units when 0 <= Messages_Minus_Instances < 2
Add 2 capacity units when 2 <= Messages_Minus_Instances < 3
Add 3 capacity units when 3 <= Messages_Minus_Instances < 4
Add 4 capacity units when 4 <= Messages_Minus_Instances < 5
Add 5 capacity units when 5 <= Messages_Minus_Instances < 6
Add 6 capacity units when 6 <= Messages_Minus_Instances < 7
Add 7 capacity units when 7 <= Messages_Minus_Instances < 8
Add 8 capacity units when 8 <= Messages_Minus_Instances < 9
Add 9 capacity units when 9 <= Messages_Minus_Instances < 10
Add 10 capacity units when 10 <= Messages_Minus_Instances < 11
Add 11 capacity units when 11 <= Messages_Minus_Instances < 12
Add 12 capacity units when 12 <= Messages_Minus_Instances < 13
Add 13 capacity units when 13 <= Messages_Minus_Instances < 14
Add 14 capacity units when 14 <= Messages_Minus_Instances < 15
Add 15 capacity units when 15 <= Messages_Minus_Instances < 16
Add 16 capacity units when 16 <= Messages_Minus_Instances < 17
Add 17 capacity units when 17 <= Messages_Minus_Instances < 18
Add 18 capacity units when 18 <= Messages_Minus_Instances < 19
Add 19 capacity units when 19 <= Messages_Minus_Instances < 20
Add 20 capacity units when 20 <= Messages_Minus_Instances < +infinity

Scale In
CloudWatch Alarm condition: q - i < -1
Actions:


Remove 1 capacity units when -1 >= Messages_minus_Instances > -2
Remove 2 capacity units when -2 >= Messages_minus_Instances > -3
Remove 3 capacity units when -3 >= Messages_minus_Instances > -4
Remove 4 capacity units when -4 >= Messages_minus_Instances > -5
Remove 5 capacity units when -5 >= Messages_minus_Instances > -6
Remove 6 capacity units when -6 >= Messages_minus_Instances > -7
Remove 7 capacity units when -7 >= Messages_minus_Instances > -8
Remove 8 capacity units when -8 >= Messages_minus_Instances > -9
Remove 9 capacity units when -9 >= Messages_minus_Instances > -10
Remove 10 capacity units when -10 >= Messages_minus_Instances > -11
Remove 11 capacity units when -11 >= Messages_minus_Instances > -12
Remove 12 capacity units when -12 >= Messages_minus_Instances > -13
Remove 13 capacity units when -13 >= Messages_minus_Instances > -14
Remove 14 capacity units when -14 >= Messages_minus_Instances > -15
Remove 15 capacity units when -15 >= Messages_minus_Instances > -16
Remove 16 capacity units when -16 >= Messages_minus_Instances > -17
Remove 17 capacity units when -17 >= Messages_minus_Instances > -18
Remove 18 capacity units when -18 >= Messages_minus_Instances > -19
Remove 19 capacity units when -19 >= Messages_minus_Instances > -20
Remove 20 capacity units when -20 >= Messages_minus_Instances > -infinity

Working

The following table encapsulates the working of AutoScaling mechanisms.

State	**q**	**i**	**q - i**	**Alarm triggered**	**Actions**
Before any message enters the queue	0	0	0	None	None
When we flood the request queue with 100 messages	100	0	100	Scale-Out Alarm Triggers	Spawn maximum of 20 or q-i number of new EC2 instances
When the number of messages between 100 and 20	20 < q < 100	20	0<q-i<80	Scale-Out alarm triggers	No new instance is spawned since the max number is 20
When the number of	0 < q < 20	20	-20 < q-i < 0	Scale-Out is in OK state Scale In alarm	Excess number of instances are killed

messages drops below 20				triggers	
When messages reach 0	0	0 < i < 20	-20 < q-i < 0	Both alarms in OK state	

2.3 Member Tasks

2.3.1 Sai Vikhyath Kudhroli (1225432689)

1. Devised an approach to integrate the image classification code with the app tier.
2. Implemented the app-tier application.
3. Implemented a method to retrieve the queue URL, and retrieve the image URL, and the unique key from the request queue and delete it on reception.
4. Modified the image classifier to directly use the image URL to classify the image without downloading the image locally, which saves enormous storage space.
5. Form a key-value pair of the image name and the classification result and push it to the output S3 bucket.
6. Map the classification result with the unique key and send this response to the response queue.
7. Write cron jobs using the no hang up (nohup) process in the web tier and app tier to start the applications automatically when the instances are booted.
8. Configure the security groups for EC2 instances and set up user policies to allow privileged access to the instances and enable SSH access and TCP connections to instances to allow requests to be sent to the web tier.
9. Configure the S3 bucket policy for the objects to be accessible to the application.

2.3.2 Gautham Maraswami (1225222063)

1. Designed methods to write to SQS write queue and read from SQS read queue.
2. Developed a Flask application to handle REST requests from workload-generator/ web browser.
3. When the controller receives the image file, the file will be uploaded to the S3 bucket using the credentials.
4. Once the file is uploaded successfully, the URL is generated for further processing.
5. Designed a UUID-based key-value pair to map the request sent from the web tier and the response received from the app tier so that even though processing at the app tier happens asynchronously, the web tier is able to return the right results.
6. Developed methodology so the messages in the response queue are accessed only once.
7. Created a web tier instance and installed all relevant packages like Flask, and Boto3 for the application to run.
8. End-to-end testing of the application was performed to make sure that there is no request loss, and no idle instances.

2.3.3 Abhilash Subhash Sanap (1225222362)

1. Set up EC2 instances for app and web tier, request and response queues, and S3 input and output buckets to start off the project.
2. Created an Amazon Machine Image (AMI) from a running app-tier EC2 instance
3. Created a launch template from the AMI created in step 2.
4. Created an AWS Autoscaling Group (ASG) using the launch template in step 3.
5. Configured the parameters of the ASG to meet the requirements as stated in the project guide.
6. Defined a custom metric **ApproximateNumberOfMessagesVisible - GroupInServiceInstances** with a math logic using AWS CloudWatch.
7. Defined an alarm each for Scale-Out and Scale In
8. Defined a Scale-Out Step Scaling policy and a Scale-In Step Scaling policy
9. Tested the setup End to End to ensure that the execution finishes well within the stipulated time of 7 minutes.

3. Testing and evaluation

3.1 Reliability Test

The application was tested to make sure the expected reliability and accuracy are achieved. The application was tested to check if it is able to handle up to one hundred requests concurrently. It is made sure all 100 requests reach the Request queue.

3.2 Scale out and Scale in Mechanism

The number of instances in the app tier is expected to linearly increase up to 20 instances. To test the above features we send 100 requests using the multi-threaded workload generator provided. The number of instances only increased to 20 even though there were more than 80 requests during the cloud watch alarm trigger. We also sent 20 requests each from two concurrently running workload generators and made sure the right response was reaching the right workload generator. The application was also tested using the postman application and also the web browser. It is also tested that when there are less than 20 requests in the queue an equivalent number of instances are active. Ie. When there were 10 messages in the queue, 12 instances were active.

3.3 Testing App Tier

Application tier is tested to make sure it is able to pick up requests from the queue, download the image from the S3 Bucket, perform image classification, and write the results to the appropriate S3 bucket. It is tested that no two app tier instances are picking up the same image. The app tier deletes the message when it has read the message from the queue. Results in the S3 Bucket are analyzed so that images are rightly classified and written.

3.4 Flow of operations test

Flow of the input image at each step from workload generator, Web tier application, request SQS Queue, app tier application S3 Bucket, response queue is tested by using appropriate print statements and count.

3.5 Output validity test
Response at the workload generator is checked against the source of truth document provided to make sure all the images are rightly classified.
The time taken for execution of one hundred images is noted as 5 minutes, the time for scale up to start 1 minute, time taken for the number of instances to reduce back to 1 active instance is 6 minutes 55 seconds.

4. Code

Technologies used: Python, Flask, Boto3, AWS EC2, AWS SQS, AWS S3, AWS Cloud watch

We developed the web application using Flask to handle REST requests, app-tier which reads and writes to a queue and executes the image classifier using python. Boto3 was used for interacting with AWS.

Files:

1. Workload generator folder: This folder contains the workload generator scripts, both multithreaded and a file that is used to test the application.
2. web_tier.py: This file contains the flask server code used for handling HTTP requests.
3. app_tier.py: This file contains the code for accessing the SQS queues and running the image classification
4. Image_classification.py: code provided for the classification of images.
5. Images-100: Folder containing 100 images used for testing
6. Setup.py: For setup of Ec2 instances, SQS Queues, and S3 buckets.

Web_tier.py

- This file contains the code for the Flask server capable of handling concurrent requests.
- Configuration is provided in a hashmap at the beginning of the file for authentication
- Upload_file_to_s3 method uploads the file to S3.
- Generate_object_url method gets the URL for the given object in S3
- Send_message_to_queue method set the message attributes and sends it to the specified queue.
- Get_sqs_url gets the URL of the queue so we can delete messages in this queue.
- Purge_message_from_queue this method deletes a particular message specified by the handle in the given queue URL.
- Receive_message_from_queue: this method is used to retrieve the response related to the given unique key
- Post_data: this method is used to intercept HTTP requests from the browser.

App_tier.py

- This file is responsible for fetching the image URLs from the request queue, invoking the image classifier with the image URL as an argument, and pushing the result to the output S3 bucket and the response queue.
- This file used Boto3 AWS SDK to perform operations on AWS IaaS resources.
- The methods of the file and their functionalities are elucidated below:
    - get_sqs_url: This method returns the URL of the queue based on the queue name which is passed as a parameter to this method.
    - receive_message_from_queue: This method retrieves messages from the queue based on the queue URL passed as the parameter. The request queue's URL is passed as the parameter to this method. The unique key and the image URL are returned from this method. This also invokes the purge_message_from_queue method after retrieving the message and the receipt handle to purge the message from the request queue.
    - purge_message_from_queue: This method takes the queue URL and the receipt handle as the parameters and deletes the message from the queue.
    - send_message_to_queue: This method takes the response queue URL, the unique key and the classification result as parameters and pushes the key and classification result into the response queue.
    - push_result_to_s3: This method takes the image name, unique key, classification label, and output bucket name as parameters. Forms a key-value pair of the image name and classification result and writes the result into the output S3 bucket.
    - listen(): This method is responsible for waiting to fetch the messages from the queue and invokes all the methods accordingly to perform the functionality of the app tier.

Set_up.ipynb

- This ipynb file contains the code to accomplish the following -
    - Create EC2 instances, check their properties, and start/stop or terminate them.
    - Create SQS queues, send/receive/read/delete the messages
    - Create S3 buckets, send data to them

4.1 Project setup and execution

Web_tier:

- Create an EC2 instance using the script or via AWS console, use ubuntu as the operating system.
- Install flask and boto3 libraries.
- Configure security and firewall for the instance so as to allow traffic only in port 5000
- Configure SSH access in the EC2 instance, and add your public key to the allowed list.
- Configure Mobaxterm in your local system so you will be able to do SCP via GUI.
- Upload web_tier.py to the EC2 instance using Mobaxterm
- Run the script using nohup

- Create cron jobs using the no hang up (nohup) process to start the applications automatically when the instances are booted.

App_tier:

- Create an EC2 instance using the script or via AWS console, using ubuntu as the operating system.
- Install Boto3 library.
- Configure the security group to allow SSH on port 22.
- Perform SSH key-gen to access the EC2 instance using SSH.
- Perform SCP to copy the app_tier.py to the EC2 instance.
- Enter the directory where the file has been copied.
- Run the script using the following command
    - nohup python3 app_tier.py > app_tier.log 2>&1 &
- Create a cron job using the below command to start the script automatically when the instances are booted.
    - @reboot nohup python3 app_tier.py > app_tier.log 2>&1 &

5. Individual Contributions – Portfolio

5.1 Sai Vikhyath Kudhroli (1225432689)

Introduction

The aim of the project is to develop an elastic application that can scale in and scale out on demand to perform image classification using AWS IaaS.

Solution

The application consists of a web tier, app tier, messaging queue service, cloud watch, and data tier that are integrated to perform image classification.

Design Contributions

- As a result of numerous discussions on different designs and architectures, technologies and services to be used, we designed a loosely coupled and modular architecture that facilitates each component of the system to perform its job independently, and the components are synchronized using the message queue services.
- My significant contribution to this project is the design and integration of the application tier into the system. I designed all the functionalities that the app tier is responsible to perform and all the integration points in the entire application.

Implementation Contributions

- Implemented the app-tier application.
- Implemented a method to retrieve the queue URL, and retrieve the image URL, and the unique key from the request queue and delete it on reception.
- Modified the image classifier to directly use the image URL to classify the image without downloading the image locally, which saves enormous storage space.
- Form a key-value pair of the image name and the classification result and push it to the output S3 bucket.
- Map the classification result with the unique key and send this response to the response queue.
- Write cron jobs using the no hang up (nohup) process in the web tier and app tier to start the applications automatically when the instances are booted.
- Configure the security groups for EC2 instances and set up user policies to allow privileged access to the instances and enable SSH access and TCP connections to instances to allow requests to be sent to the web tier.
- Configure the S3 bucket policy for the objects to be accessible to the application.

Testing Contributions

I was responsible for testing the individual modules of the app tier, the integration of the app tier into the application, and testing the end-to-end system with varying loads.

New Skills and Technologies Acquired

- Using AWS EC2, AWS S3, AWS SQS, AWS Cloud Watch, and Boto3 AWS SDK to develop an elastic, on-demand autoscaling application and deployment on cloud resources.
- Configuring Security groups to enable communication between EC2 instances and other devices, set up bucket policies and access control lists. Creation of various autoscaling policies.

5.2 Gautham Maraswami

Task: My responsibilities included the development of servers to handle requests from the workload generator/ API and create mechanisms for interaction with the App tier.

Deciding on the technologies: Individually experimented with the boto3 library and after analyzing the documentation, wrote example scripts for the creation of an S3 bucket, uploading and downloading files from S3, SQS queue creation, inserting and deleting messages in SQS queues Evaluated different technologies like Spring Boot-tomcat, and Flask-nginx for server development and decided to use Flask-Python for server-side development. After studying documentation, I decided to use Boto3 for AWS-related operations in python.

Application Development: The application development was divided into the following steps. Each step was developed and tested individually and as an Integration.

1. Design Rest API Controller: An endpoint was developed which takes the file as a request parameter and stores the file in the instance locally.
2. Upload to S3: In this step, the file received at the controller is loaded to the S3 bucket using the S3 Api which was experimented with previously and generated the URL for the same
    a. Experimented on various flags including accessing objects using signed URL, Access control List, etc.
3. Create SQS Message: Created a UUID and added it to the message attributes section of the message, the message body contains the image URL received from the S3.
    a. The UUID sent will help in identifying the correct response from the response queue
4. Thread to read the message from the response queue: Here in an infinite while loop we check the response queue for the messages. We create a map of UUID vs response.
5. Return result: the code checks the thread-safe map populated in the previous step and returns the response to the rest call. If the value is not present for the current UUID the code waits for the map to be populated.

Understanding other Sections: Spent time with other developers to understand the app tier code, Autoscaling mechanisms, and cloud watch metrics.

Challenges Faced: As a new developer in python, flask, and Boto3, faced many challenges regarding syntax and usage which were resolved using the relevant documentation. The server timed out for requests with a wait time of more than 6 minutes, to address this issue, tried re-implementing the entire code base in Spring Boot and deployed it on the Tomcat server.

Learning from the project

Ability to develop an application and deploy it on EC2 Instance, Ability to develop APIS to interact with S3, SQS. Develop an auto-scaling mechanism using various metrics like CPU usage, memory, number of messages in the queue, etc. Ability to create an image and deploy instances using the created images.

5.3 Abhilash Subhash Sanap

Responsibilities

- Set up EC2 instances for app and web tier, request and response queues, and S3 input and output buckets to start off the project.
- Implement AutoScaling to scale out and scale in based on the size of the request queue.
- Ensure that the application runs within 7 minutes.

Contributions

Setting Up

- I read the boto3 documentation and set up the EC2 instances for the web and app tier.
- I also created the request and response queues and S3 buckets that hold the images and results.

AutoScaling

- After the app tier was ready, I created an Amazon Machine Image (AMI) from it configuring the correct parameters.
- Next, I created a Launch Template from the saved AMI. The AutoScaling Group (ASG) I created next used the same Launch Template with a minimum capacity of 1 and the maximum capacity of 20.
- I studied the relevance of Simple, Step, and Target tracking to the problem at hand and chose Step Scaling after much deliberation.
- The requirement was to scale as many new EC2 instances as there are unread messages in the request queue. Hence I decided to use the metric - **ApproximateNumberOfMessagesVisible (request queue) - GroupInServiceInstances (ASG).**
- The same metric is used with two separate alarms to trigger actions that enable Scale Out and Scale In of EC2 instances. The alarm condition for Scale-Out is the metric value greater than 1 and the alarm for Scale-In is the metric value lesser than 0. With the aforementioned configuration, the application runs as expected - scaling up as messages flood the request queue, serving all the results, and scaling down as the number of unprocessed messages drops well within seven minutes.
- Tested the application end-to-end multiple times with different loads to ensure that it works as expected.

Major Issues Faced

Target Tracking Scaling - As we started with Target tracking scaling, although it did scale up as expected, the scale-down operation took too long to work as AWS has the default value of 15 data points in 15 minutes for scale in operation. I explored all workarounds but could not change this policy. Hence, we moved to Step Scaling with 20 rules each in Scale Out, and Scale In.

Learning Outcomes

- Using Infrastructure as a Service modules like AWS EC2, AWS SQS, AWS Cloud Watch, AWS S3, and AWS AutoScaling Group
- Learning AWS CLI and boto3 AWS SDK to develop applications based on AWS
- In-depth understanding of AWS AutoScaling Groups, CloudWatch alarms and Scaling policies like Step and Target Tracking.