

CSE 546 - Project Report

Deval Pandya - 1225424200

Karthik Ravi Kumar - 1225910467

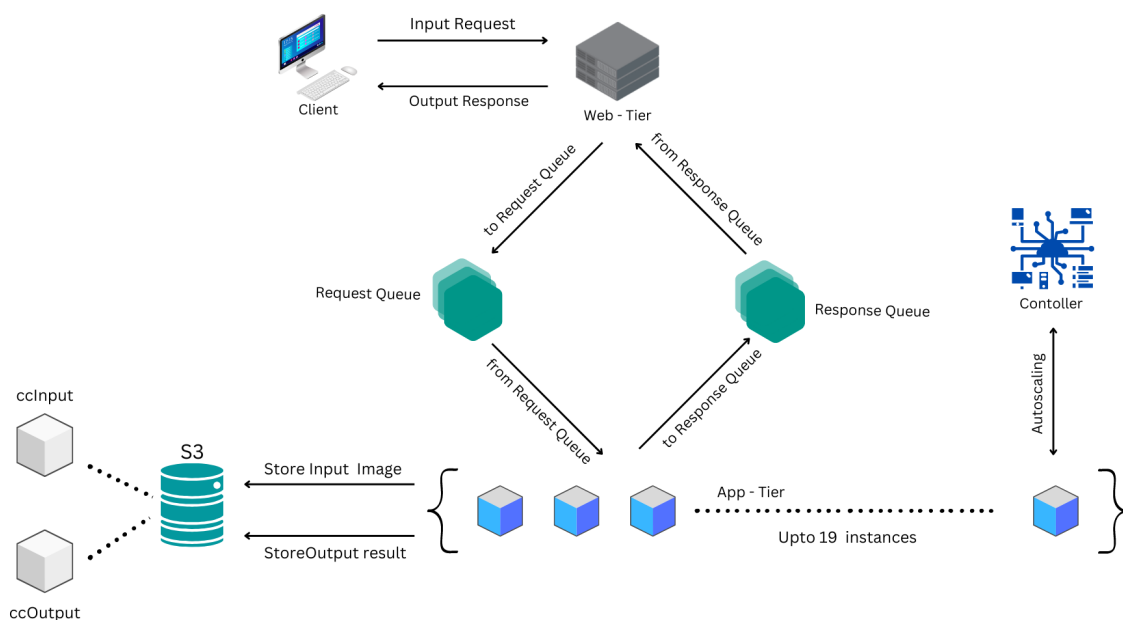
Tirth Hihoriya - 1225413475


1. Problem Statement

To build a scalable cloud web app using AWS resources that recognizes images based on a given classifying model. The processing cloud instances scale up and down based on the inflow of requests. The app uses AWS EC2, AWS SQS, and AWS S3 services. The problem that we are solving is that the auto-scaling implemented can process higher loads of requests by optimizing the number of instances according to the need. This problem is important for us to solve as it helps us in understanding the key concepts of cloud computing which are resource pooling, on-demand self-service, elasticity, and measured service

2. Design and implementation

2.1 Architecture





From the high-level architecture diagram, we can see that the web application consists of three main components - The web tier, the Controller, and the App tier.

Web tier - The Web tier is used to receive the request with image input from the user, encode it and upload it to the SQS RequestQueue with image attributes for image processing. Once processing is done by the app tier, the web tier is used to retrieve the result from the output SQS queue and display the output in the user interface.

Controller - The second module in our system is the controller. The controller keeps track of the number of incoming requests in the RequestQueue and manages the number of instances present accordingly. The controller deploys up to 20 app tier instances based on our auto-scaling algorithm

App tier - The app tier is the processing component of the system. The app tier also consists of a listener which picks up input requests from the RequestQueue, decodes them, and sends them to the image processing model. The output is added to the ResponseQueue and is then both the inputs and outputs saved in the S3 buckets. Once this is done, it deletes the message from the RequestQueue.

SQS queues - there are two queues, RequestQueue and ResponseQueue. They are used to deliver messages between the App tier and the web tier. The app tier takes care of deleting the messages in the RequestQueue once the input has been received. The queues have long polling enabled with ReceiveMessageWaitTime of 1 second.

S3 Buckets - the S3 buckets (cc-project-input and cc-project-output) are used to store the inputs and the outputs after processing are done by the app tier.

Initially, we have **zero** app tiers and **one** web tier instance running. These app tier instances will be scaled in and out according to the requests loaded from the web tier. The Auto Scaling algorithm in the controller is used to decide whether additional app tier instances are to be created or terminated. Once the model processes the image input and gives out the resulting output, it is fed into the output SQS queue and both input and output are stored in s3 buckets, the web tier then receives the output from the ResponseQueue and displays it in the user interface.



2.2 Autoscaling

We have implemented the autoscaling logic in our controller which runs in a separate EC2 instance. The controller is used to scale out or scale in the App Tier instances based on the number of requests remaining for processing in the Request Queue. Controller finds out the queue length by continuously querying the Queues at a fixed time interval.

2.2.a Upscaling

When the controller finds that the total number of requests in the Request Queue exceeds the total number of App Tier EC2 Instances running or starting up, it will Scale Up to start more instances of App Tier (up to a maximum of 20 instances). For example, if the Request Queue has 15 messages remaining and there are 7 App Tier instances running, the controller will launch another 8 instances of App Tier to meet the demand.

2.2.b Downscaling

When the controller finds that the total number of requests in the Request Queue are less than the total number of App Tier EC2 Instances running or starting up, it will Scale Down to terminate the instances of App Tier. But before terminating any instance, the controller waits for about 20 seconds during which it will poll the Request Queue about 4 more times. During that interval, if the number of requests goes back higher than the number of App Tier instances running, it will give up on terminating those instances. For example, if the Request Queue has 7 messages remaining and there are 15 App Tier instances running, the controller will terminate 8 instances of App Tier, after waiting for 20 seconds, to meet the demand. The waiting period is to overcome the issue of SQS returning an incorrect number of messages sometimes.

2.3 Member Tasks

Deval Pandya - (ASU ID: 1225424200)

I have designed the end to end flow of this project, like determining the AWS components to be used and setting them up like SQS Queues, S3 Buckets, Security Groups, EC2 instances and their user data, IAM Instance profiles. I also played a part in designing the logic for Web Tier, App Tier, and Controller. I implemented the code for the controller which is used for Autoscaling of App Tier instances as per the number of messages in the Request Queue. I did the testing for the whole application which involved sending several requests (Single/Concurrent) from the workload generators provided to us and optimized the algorithms of App Tier and Web Tier in order to showcase the features of autoscaling in our project.

Karthik – (ASU ID: 1225910467).

I have designed the Web tier. The listener part of the web tier includes receiving messages from the user, storing it and encoding the images. Other parts include setting up and sending messages to the request queue, testing the request queue, receiving messages from the response queue and printing the output.

Tirth Hihoriya - (ASU ID: 1225413475).

I have designed the App-Tier. The app-tier includes basic image classification program. It receives a task from RequestQueue and processes it. Once it is done with processing, it sends the output to the Response Queue. It also stores the input in input-bucket and output in output-bucket. If no new task comes to it, it waits for 5 seconds. I have tested the app-tier code rigorously and updated it to have more robust code.

3. Testing and evaluation

3.a Testing

Following scenarios were tested thoroughly for autoscaling of the project. For testing these scenarios we uploaded a different number of images at different times during the processing of the project:

RequestQueue Length	Active Instances	Action
0	2	The controller terminates the 2 instances after waiting for 20 seconds.
5	0	Controller launches 5 instances
5	3	Controller launches 2 instances
5	5	Controller takes no action
50	0	Controller launches 20 instances
100	20	Controller terminates 10 instances after waiting for 20 seconds
0	20	Controller terminates 20 instances after waiting for 20 seconds
changes from 100 to 50	20	Controller takes no action



3.b Evaluation

We noted down the timings needed for various actions in the project:

Total Time taken to process and reply with 100 results: **About 6 minutes**

Time taken after launch of first App Tier instance to start the processing of Request Queue: **about 2 minutes**

3.c Video for full working project

https://drive.google.com/file/d/1Qw1KlTcVgxFKJgd8L_aqsYG3gE5W_pAC/view?usp=sharing

Login with your ASU id to see it.

3.d Challenges Faced and solutions

- Inconsistent reporting of Queue Length: we solved this by configuring the queue to long polling which reduced the number of empty ReceiveMessage Functions. Moreover, we introduced a threshold number before which autoscaling would not be performed.
- Clash in receiving a message from ResponseQueue: We solved this problem by querying the ResponseQueue at random times between 1 second to 4 seconds and keeping the Visibility Timeout to 2 seconds which gave enough time to each web server thread to find its respective result.

4. Code

4.1 Functionality of every program

Web.py :

In our web tier code, first we import the required libraries (Flask , boto3, base64 etc), setup the required directories and initialize the client for the SQS queues. We use Flask library to receive the POST requests with the images and then store the images in a local directory. The images are then encoded using base64 and then added to the SQS RequestQueue. The Second functionality of the web-tier is to receive messages from the ResponseQueue using a 'while True' loop and then display it in the user Interface. If the multithreaded workload generator is used, each request is treated independently in Flask.


Controller.py :

The controller uses a Python3 program to monitor the Request Queue at fixed time intervals of 5 seconds and Scale Up/Down the App Tier instances for processing the fluctuating number of requests. It uses the following logic to do the autoscaling:

- If the number of messages in the Request Queue (Available + In Flight) is greater than the number of App Tier instances (Running/Pending), the controller launches 1 App Tier instance per each message to meet the increased demand.
- If the number of messages in the Request Queue (Available + In Flight) is lesser than the number of App Tier instances(Running/Pending), the controller scales down the number of App Tier instances by terminating them until they match the number of requests.
- In order to overcome inconsistencies in the reporting of the number of messages by SQS, the controller waits for 20 seconds (4 polls to SQS at 5 seconds each) before terminating any excessive App Tier instances running. During this interval, if the number of requests rises back up, the controller does not terminate the instances or only terminates lesser instances if the requests are still less than the Running instances.

App.py :

In the beginning of the code, the necessary global variables and objects are initialized, including the SQS queues and S3 buckets using the boto3 library in python. The "while True" loop is created which does the majority of the task in app-tier, starting with checking for messages in the RequestQueue. When a message is received, it decodes the message using base64 library because it was earlier encoded by web-tier before sending it to RequestQueue. Then the app-tier sends it to the image classification model. Once the classification is complete, it sends the output to



ResponseQueue. I also store the input image in the input bucket. Afterwards, the program stores output results along with the image file name in the output-bucket. As all these are in an infinite while loop, it repeats the same for the next input image. If there are no more messages, the instance waits for 5 seconds and after 5 seconds again checks for input in the RequestQueue. In the code there are many points where there is a chance of exceptions. These exceptions may occur due to reasons such as, the code might not get a message from RequestQueue, the input and output might not be stored in S3 buckets successfully, the message from RequestQueue might not be deleted successfully after it gets processed, etc. Thus, to handle it the code is taken care of with the 'try-except' block. If any exception occurs then it will wait for 5 seconds and will try to process again.

4.2 Installation and execution

4.2.a Web Tier

Launch an EC2 ubuntu instance with following properties:

Key Pair: Whichever necessary (In our case KP1 is .ppk and KP2 is .pem)

Security Group: Allow-All-Traffic

Iam instance profile: EC2-SQS-S3-FullAccess

Userdata:

```
#cloud-boothook
```

```
#!/bin/bash
```

```
sudo apt update
```


```
sudo apt install -y python3
```

```
sudo apt install -y python3-flask
```

```
sudo apt install -y python3-boto3
```

```
sudo apt install -y tmux
```

```
sudo apt install -y awscli
```

```
mkdir /home/ubuntu/.aws  
aws s3 cp s3://cc-project-extra/config /home/ubuntu/.aws/  
aws s3 cp s3://cc-project-extra/config ~/.aws/  
aws s3 cp s3://cc-project-extra/web.py /home/ubuntu/  
mkdir /home/ubuntu/savedImages  
chmod +777 /home/ubuntu/savedImages  
chmod -R 777 /home/ubuntu
```

Connect to the instance and use command 'python3 web.py' to run web server

4.2.b Controller

Launch an EC2 ubuntu instance with following properties:


Key Pair: Whichever necessary (In our case KP1 is .ppk and KP2 is .pem)

Security Group: Allow-All-Traffic

Iam instance profile: EC2-SQS-S3-FullAccess

Userdata:

```
#cloud-boothook  
#!/bin/bash  
sudo apt update  
sudo apt install -y python3  
sudo apt install -y python3-flask  
sudo apt install -y python3-boto3  
sudo apt install -y tmux  
sudo apt install -y awscli  
mkdir /home/ubuntu/.aws
```



```
aws s3 cp s3://cc-project-extra/config /home/ubuntu/.aws/
```

```
aws s3 cp s3://cc-project-extra/config ~/.aws/
```

```
aws s3 cp s3://cc-project-extra/controller.py /home/ubuntu/
```

```
mkdir /home/ubuntu/savedImages
```

```
chmod +777 /home/ubuntu/savedImages
```

```
chmod -R 777 /home/ubuntu
```

Connect to the instance and use command 'python3 controller.py' to run web server

4.3.c Execution

Use the web tier url to send the requests using workload generator:

http://<Web_Tier_Public_Address>:8080/upload

This will add requests to sqs which in turn will trigger the controller to start up App Tier instances.