CSE 546 - Project Report

Akshaya Kumar - 1217106670
Srihari Thangirala - 1219435815
Venkatramanan Srinivasan - 1217023522

1. Problem statement

To build a distributed elastic computing system that makes use of IAAS resources from AWS to implement an image recognition application that scales up and down based on demand.

2. Design and implementation

2.1 Architecture

2.1.1 Design Summary:

As seen from the high-level architecture diagram (Figure 1), the overall components of this system include three major components - The Web tier, the Controller, and the App tier. The Web tier is responsible to upload the image requests and put them in the input SQS queue and also in an S3 bucket for persistence. The controller does the job of assessing the length of the input SQS queue and accordingly creates EC2 instances based on the load. A maximum of 19 app instances will be created to process all the images uploaded. The controller and web tier are run from the same EC2 instance. When this web-tier EC2 instance is started, the web tier and controller will automatically start running and the user can upload the images directly using the public IP. The app service replicated in all the app tiers created by the controller is responsible to take the image details from the input SQS queue, processing it and place the output in the output SQS queue. The app tier also places the result in an S3 bucket for persistence. The Web tier then takes the result from the output SQS queue and shows it as user output.

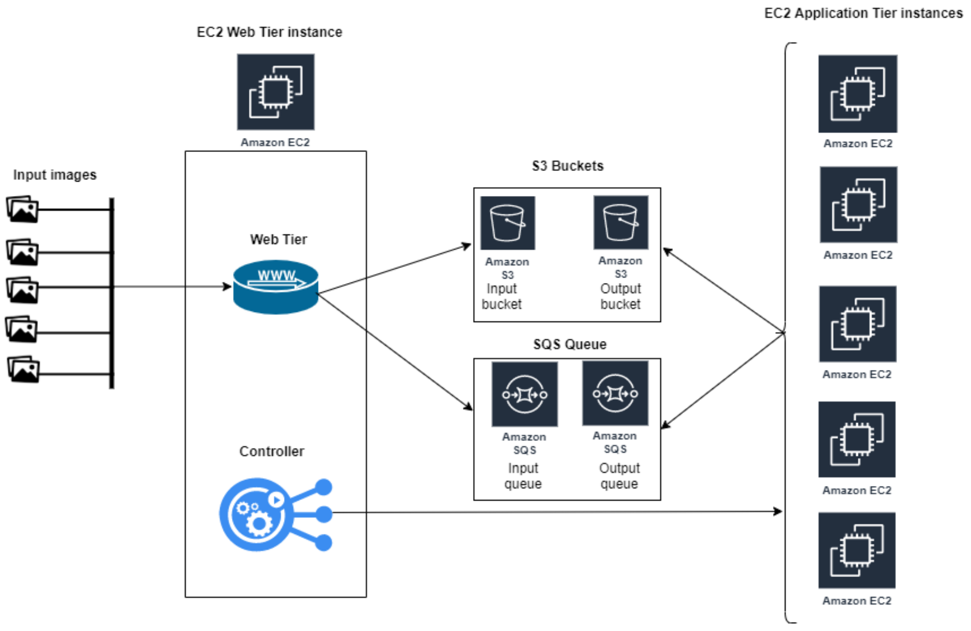


Figure 1: High Level architecture

2.1.2 AWS Components used:

a. **EC2:**
The Elastic Cloud Compute is the virtual server environment where we have hosted web tier, controller and the app services

b. **SQS:**
The Simple Queue Service is utilized for communication between the web tier and app tier. SQS queue is used both for input and output queue.

c. **S3:**
The Simple Storage service of AWS is used to store items that need persistence. The input images and the output image classification results are stored in separate S3 buckets.

d. **IAM Role:**
IAM user role is created with specific permissions and used to access the AWS resources

e. **AMI**
The given custom Amazon Machine Image with image recognition model is further customized to include app service to create the multiple app tiers.

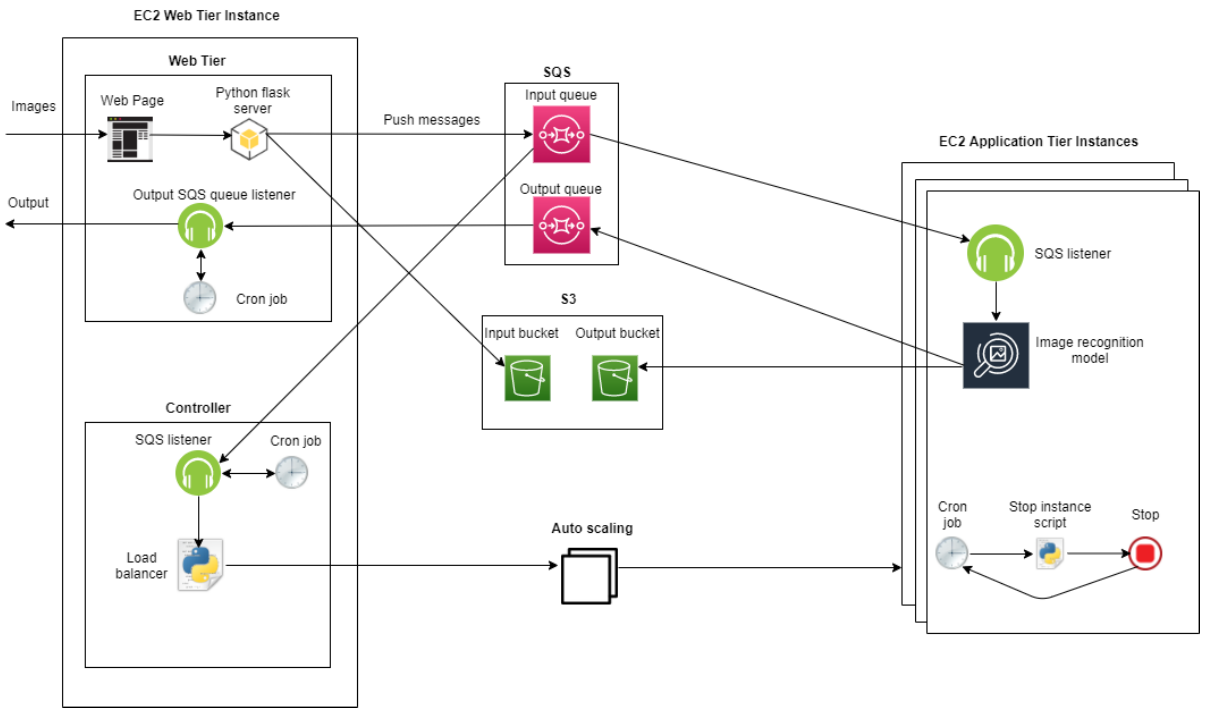


Figure 2: Detailed Architecture

2.1.3 Implementation components

a. **Web tier**
As shown in the detailed architecture(Figure 2) the web-tier is the interface for the user to upload images. Flask server is used to run the web service. There are three main functions performed by the web tier. First it takes all the images uploaded by the user. After taking the uploaded files from user interface it puts all the images in input S3 bucket.It also adds the messages into input SQS queue with the image attributes like image name and URL.After all the processing is done it retrieves all the images from output SQS and displays the results in the user interface.

b. Controller

The controller is also included in the same EC2 instance as that of the Web tier. The cron job scheduler will start the controller component inside this instance. The controller is a simple Python3 SQS listener. The controller also has the ability to spawn new instances with the given AMI id, and perform operations based on states of other instances that are running. The controller polls the Input SQS in a repeated fashion to check for messages that have been placed there by the Web tier for processing. Based on the number of messages, the Controller will create/start EC2 instances (App tier) that aid in the processing of the messages parallelly to increase the speed of computation.

Controller Logic:

- Queue length is 0?
    - Continue to poll the input queue for new messages.
- Queue length is greater than maximum threshold (19 instances)?
    - If there are any idle instances, then create the remaining number of instances not exceeding a total of 19 along with idle instances and then start all the idle instances.
    - If there are no idle instances, create 19 instances.
    - If there are only idle instances, start all of them.
- Queue length is greater than the number of idle instances but less than the maximum threshold?
    - Create the required number of instances not exceeding the maximum threshold and start all the idle instances.
    - If the number of new instances have to be created and the number of idle instances and the active instances sum up to greater than the maximum threshold, create the maximum instances you can spawn under the threshold and start all the idle instances
- Queue length is less than the number of idle instances?
    - Start queue length number of instances.

c. App tier

As seen from the detailed architecture (Figure 2), the app tier consists of two components: One is the core SQS Listener that waits for the input queue to receive messages and call the image recognition model on it. The results of the image recognition are stored in S3 output bucket for persistence and fed into an SQS queue for consumption by the SQS listener of the Web tier. The other component is the cron job daemon that runs the stop script. This keeps looking out for the idle time of the instance and automatically stops the instance if there are no messages in the queue.

2.2 Autoscaling

Auto scaling is a mark of a perfectly elastic application. This ensures that the resources are scaled up and down depending on the incoming traffic, thus minimizing resource usage. This application performs auto scaling in the following ways:

2.2.1 Auto Scale up

The controller part is responsible for the upscaling of the application. The controller acts as the listener that will continuously poll the Input SQS for messages. If the queue length is greater than 1, the controller will check the number of currently active instances. The controller will now calculate the number of instances it needs to spawn in addition to the active instances (which are busy and are processing images currently) within the threshold (20 instances - as a part of AWS free tier). It should be noted that the controller will give priority to start the available instances that are idle (‘*stopped’* state) over creating new instances. The controller will keep polling the SQS in a consistent manner to make sure that even the new messages that may be added by the Web tier are also processed.

2.2.2 Auto Scale down

The scale down is taken care of by the app instances themselves. If there are no more messages to be consumed in the input queue, the queue length becomes 0. In this case a background daemon running in each app instance checks this queue length and the number of processes running in that instance. If this daemon finds the instance to be idle for more than 10 seconds, then it automatically stops the instance. Thus, the scale down part, instead of being controlled by the controller, is being taken care of by each app instance individually.

3. Testing and evaluation

3.1 Component level unit testing

a. Web tier

The following functionalities were tested while developing Web service:

- Multiple image upload at once.
- Not allowing the user to upload without selecting at least one image.
- Storing the images in the input S3 bucket.
- Sending messages to input SQS queue with image attributes.
- Retrieving the messages from output SQS queue.
- Verifying the displayed results match with the ones that are in the output S3 bucket.

b. Controller

The following functionalities were tested while developing the Controller:

- Checking for states of the instances that are currently spawned.
- Create one or more instances for a given AMI id.
- Start instances that are in ‘*stopped’* state.
- Get the count of active/idle instances.
- Getting the count of the number of messages in the queue
- Implementation of the waiter function (wait for instances to get to running state)
- Based on the number of messages in the queue and the number of currently active instances, check if the correct number of instances are started or spawned (Unit testing was performed using hardcoded values of queue length).

c. App tier

The following functionalities were tested while developing app service:

- Receive messages from SQS queue - Long polling and short polling.
- Downloading items from S3 bucket.
- Verified results from the image classification model using various images.
- Sending messages to SQS queue with calculated results
- Uploading the computed image classification results onto S3 bucket

3.2 Integration Testing

This step follows the unit testing of all the components individually. From the word integration we can understand that this testing is performed by combining the individual components and perform end to end testing and make sure everything works in a proper flow. The following were tested:

- Upload images in the Web tier and made sure that the necessary fields were input in the Input SQS. In addition to that it was also made sure that the images are placed in the input S3 bucket.
- Parallely the controller was made to poll the Input SQS at a certain frequency to check for messages. During this phase, the controller was made to run indefinitely for the polling purpose as opposed to the unit testing where the code was run in single iterations to test all the edge cases alone. The controller was also tested for spawning of all the EC2 app tier instances based on the load that is received from the Web tier which is noted from the Input SQS.
- As and when the app tier instances get spawned, a cron job runs a python program to consume messages from the queue and process the image and also place the output in the Output SQS and also the Output S3 bucket.
- Final portion of the testing included retrieving the output from the Output SQS and S3 bucket and displaying it on the webpage.

3.3 Application level Load testing

The following are the cases for which the application was tested:

Case	Queue length	Instances	Output
1	100	Active: 0 Idle: 0	19 instances were created. All the instances were active until the images were processed and were shut down after a period of inactivity on the Input SQS
2	100	Active: 0 Idle: 19	19 idle instances were started. All the instances were active until the images were processed and were shut down after a period of inactivity on the Input SQS
3	6	Active: 10 Idle: 9	No new instances were created, 6 idle instances were started. 16 started instances were active until the images were processed and were shut down after a period of inactivity on the Input SQS

4	12	Active: 10 Idle : 9	The 9 idle instances were started. All the instances were active until the images were processed and were shut down after a period of inactivity on the Input SQS
5	8	Active: 0 Idle: 5	The 5 idle instances were started, and 3 new instances were created. 8 instances were active until the images were processed and were shut down after a period of inactivity on the Input SQS
6	8	Active : 6 Idle : 5	The 5 idle instances were started, and 3 new instances were created. 14 instances were active until the images were processed and were shut down after a period of inactivity on the Input SQS
7	20	Active : 6 Idle : 5	The 5 idle instances were started, and 8 new instances were created. 19 instances were active until the images were processed and were shut down after a period of inactivity on the Input SQS
8	1	Active : 8 Idle : 3	1 idle instance was started. 9 instances were active until the images were processed and were shut down after a period of inactivity on the Input SQS

For the above test cases, before each test case, we fed the right amount of images to the application in order for the application to reach the desired test case and then we fed the number of messages mentioned in the 'Queue length' column. The above test cases were not only run in single iteration but also tested in different permutations. Rigorous iterations of the above were performed to ensure that the application does not hang or break at any point in time.

Challenges faced during the testing phase:

- More edge cases and erroneous scenarios occurred during the testing. This was solved by refactoring and tweaking the code to handle those cases.
- Fluctuations in queue length read. To solve this, we retrieve the queue length upto 50 times and find the mode of those values to give a more accurate value.
- Starting less instances than required because of minor delays in the time it takes for the instance state to update to *pending.* Solved this by placing a waiter function that waits for the instance to get to *running* state.
- Varying boot times of EC2 instances due to limitations of AWS free tier.

4. Code

App.py

This contains all the components for the Web-Tier. Flask framework is used to build the web interface and also communicate with S3 and SQS. The program takes the files that are uploaded by the user, stores them in the input S3 bucket and also adds messages to the input SQS queue input-queue.fifo with image attributes. After all the processing is done it retrieves the messages which are classification results from output queue output-queue.fifo which are displayed in the user interface.

Image_upload.html
This is the main file for the user interface. It has all the html blocks and the styling required for the file upload view. In the view it has three buttons, one for choosing files to upload, one for uploading the images and one for showing the results after processing.

Controller.py
This is the file that is essential to spawn EC2 instances to process the images. The python program acts as a listener to the input SQS and creates/starts the EC2 instances using the AMI id of the APP tier necessary for processing the images. This controller is in the same instance as the Web tier. This instance is started by a cron job at the start of the Web tier.

EC2.py
This is the main service to process the input images. This python program acts as a listener to the input queue input-queue.fifo and processes messages as and when they get pushed onto this queue. Once a message is received in the queue, it extracts the image name and downloads the image from the input S3 bucket. Once extracted the image, the image_classification.py is run which produces a classification label for the input image. This output is then placed on the output SQS queue to be consumed by the web tier. This is also placed in the S3 bucket for persistence. Once processing is done, the message is deleted from the input queue. If there are any failures during the course of this processing, the message is returned back to the input queue and it will be made available to other listeners to be consumed.

Stop_instance.py
This python script runs on every app instance along with the app service and monitors if the image recognition process is run. Even after all messages are consumed from the queue, we do not want the app instances to keep running. So this script is written to monitor the process in a time interval of 10 seconds. If there is no process running and if the input queue length is 0, then it waits for another 10 seconds before initiating auto shut down. This script essentially aids in scaling down of the elastic application.

Installation and Running:

Our design runs the controller and the web tier from the same app instance.

Python packages installation:

Install the following packages into an EC2 instance:

1) Install required python version if not available - for this project python 3.8 is used.
2) Install pip package manager for installing specific python packages required.
3) Install flask using pip which is used to run the web tier.
4) Install paramiko using pip which is used to ssh into the instance if needed.
5) Install Boto3 using pip - python package for interaction with all AWS services.
6) Install AWS CLI in the EC2 instance to setup aws credentials and authenticate.
7) Install apache2 and libapache2-mod-wsgi packages to run the flask webserver.

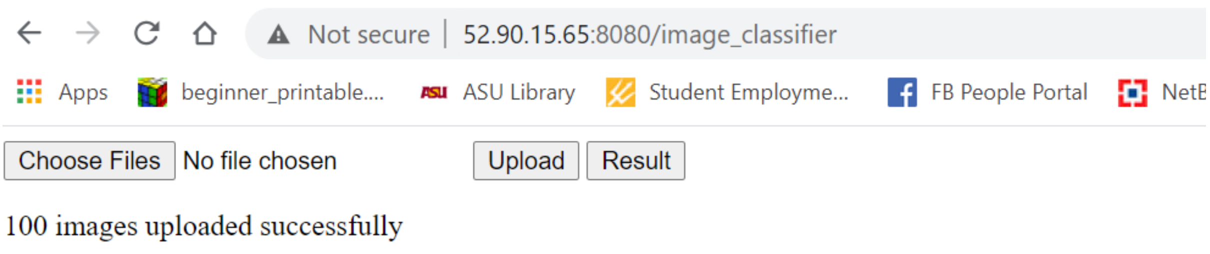
Setup of web-tier and controller EC2:

1) Add app.py to /home/ubuntu/webtier in the web-tier instance which is the main file for the web service to run and also add image_upload.html to /home/ubuntu/webtier/templates which is the main file for user interface.
2) Add controller.py to /home/ubuntu/webtier. This is the main file to run the controller. A copy of the pem file has to be in the same directory as the controller.py.
3) We have added cron jobs for both web tier (app.py) and controller (controller.py) to start when the EC2 instance boots up. We can run it manually using **python3 app.py** to run the web server and **python3 controller.py** to start the controller.
4) Once the images are uploaded, the app tiers will automatically get created by the controller with the AMI **ami-0ebf936739e9dc421.** This is the custom AMI created by adding image processing logic to the provided AMI.
5) Once web tier and controller are running, the images can be upload through a UI that can be accessed using the URL: http://52.90.15.65:8080/image_classifier
6) After choosing the files, the controller will start picking up the images after clicking the Upload button.
7) After the entire processing is done and the results are seen in the output bucket and output queue, the results can be fetched on to the UI by clicking on the Results button.

Results:

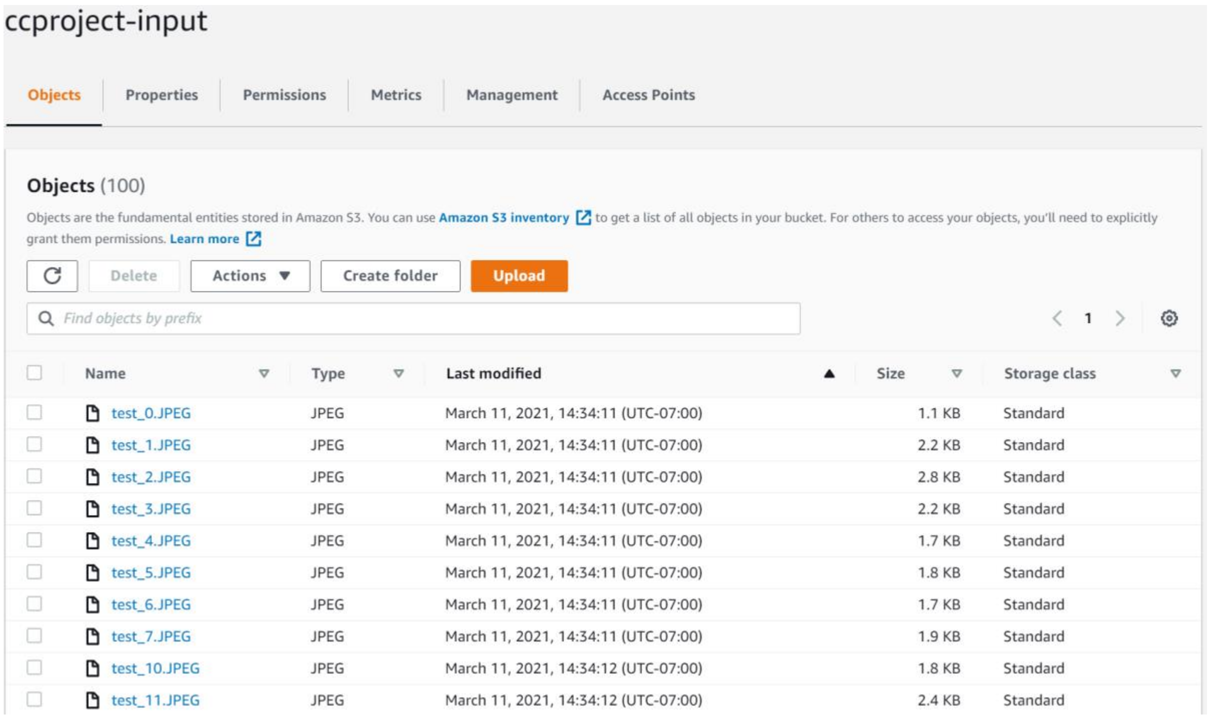
While testing, we found that the EC2 start/creation took a longer latency during peak times. The system's best latency is as follows:

- End to end latency of **4.30 minutes for 100 images**
- End to end latency of **5.01 minutes for 150 images**
- End to end latency of **6.37 minutes for 200 images**



100 images uploaded successfully

Figure 3: Upload of 100 images

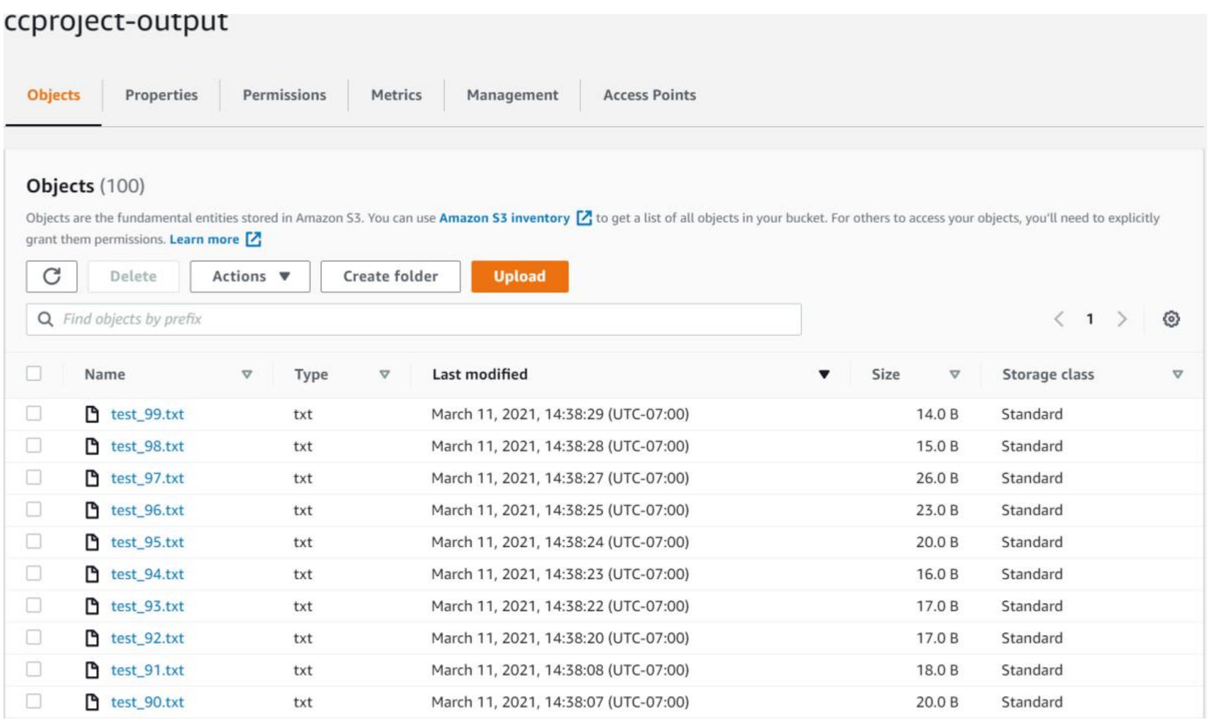


Objects (100)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
test_0.JPEG	JPEG	March 11, 2021, 14:34:11 (UTC-07:00)	1.1 KB	Standard
test_1.JPEG	JPEG	March 11, 2021, 14:34:11 (UTC-07:00)	2.2 KB	Standard
test_2.JPEG	JPEG	March 11, 2021, 14:34:11 (UTC-07:00)	2.8 KB	Standard
test_3.JPEG	JPEG	March 11, 2021, 14:34:11 (UTC-07:00)	2.2 KB	Standard
test_4.JPEG	JPEG	March 11, 2021, 14:34:11 (UTC-07:00)	1.7 KB	Standard
test_5.JPEG	JPEG	March 11, 2021, 14:34:11 (UTC-07:00)	1.8 KB	Standard
test_6.JPEG	JPEG	March 11, 2021, 14:34:11 (UTC-07:00)	1.7 KB	Standard
test_7.JPEG	JPEG	March 11, 2021, 14:34:11 (UTC-07:00)	1.9 KB	Standard
test_10.JPEG	JPEG	March 11, 2021, 14:34:12 (UTC-07:00)	1.8 KB	Standard
test_11.JPEG	JPEG	March 11, 2021, 14:34:12 (UTC-07:00)	2.4 KB	Standard

Figure 4: S3 Input images bucket (100 images)



Objects (100)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
test_99.txt	txt	March 11, 2021, 14:38:29 (UTC-07:00)	14.0 B	Standard
test_98.txt	txt	March 11, 2021, 14:38:28 (UTC-07:00)	15.0 B	Standard
test_97.txt	txt	March 11, 2021, 14:38:27 (UTC-07:00)	26.0 B	Standard
test_96.txt	txt	March 11, 2021, 14:38:25 (UTC-07:00)	23.0 B	Standard
test_95.txt	txt	March 11, 2021, 14:38:24 (UTC-07:00)	20.0 B	Standard
test_94.txt	txt	March 11, 2021, 14:38:23 (UTC-07:00)	16.0 B	Standard
test_93.txt	txt	March 11, 2021, 14:38:22 (UTC-07:00)	17.0 B	Standard
test_92.txt	txt	March 11, 2021, 14:38:20 (UTC-07:00)	17.0 B	Standard
test_91.txt	txt	March 11, 2021, 14:38:08 (UTC-07:00)	18.0 B	Standard
test_90.txt	txt	March 11, 2021, 14:38:07 (UTC-07:00)	20.0 B	Standard

Figure 5: S3 Output results bucket (100 images)

5. Individual contributions (optional)

Srihari Thangirala - MCS - 1219435815

Design:

I was involved in the end-to-end design of this project. During the requirements gathering phase, I reviewed AWS Components to use for this project, gathered knowledge about AWS Python SDK and boto3 python library to work with different services and components of AWS. I chose to go with the python boto3 library since it is very lightweight and provided easy to implement AWS functionalities. I worked on the web-tier, building the user interface, connecting the web-tier with other components like S3 and SQS to finish the end-to-end flow of the project.

Implementation:

I implemented the web-tier that has three main functions. I developed a Web-tier using flask framework in the backend and HTML, CSS, Javascript in the frontend. The implementation of the backend is done in the flask as it is very lightweight and faster in execution. The user interface has an image select input using which the user can upload one / multiple images. Once the user clicks on the upload button the images are fetched in the backend using flask request methods. The images are then uploaded into the input S3 bucket using the functions from the boto3 library to connect to the S3 service. Once an image is uploaded into the s3 bucket the image attributes are sent as a message to the SQS input queue using boto3 library functions to connect to SQS. After all the images are uploaded the user is provided with a message that all the images are uploaded. Once the images are processed by the app-tier and the results are added as messages in output on the click of the result button in the user interface the messages are fetched from the output SQS queue and are shown to the user in the user interface.

Testing:

I worked on unit testing of the web tier and the corresponding components that are connected to the web tier in the flow of the application. Tested the image upload functionality with one and multiple images (1,10,50,100,150,200). Testing the user is provided with appropriate messages when the user uploads the images and also when the user tries to upload without selecting at least one image. I have tested the backend of the web tier whether all the images are uploaded to the S3 bucket and also messages are sent to the SQS input queue. I have tested the results from the output SQS queue and checked whether they are matching with actual classification results. We did end-to-end testing by going through all possible test cases. I collaborated extensively in load testing the entire application by sending different amounts of requests to process and checking all the components are working as expected.

Venkatramanan Srinivasan - 1217023522

Design:
I was involved in the initial design phase of the project. During this phase in the project, I learnt about the multiple AWS offerings in the free tier and helped decide the way we can use EC2 instances, S3 bucket and SQS for the project. I was responsible for implementing the controller in this application. During the initial phase of my development, I chose the controller to act as a load balancer which will monitor both the SQS on one side and also the load on each of the EC2 instances on the other and manage the spawning of the instances accordingly. Later stages of development I had come to understand that this made the controller stateful since it stores the ssh connection details along with other instance related information. This also led to a single point of failure in the machine. Due to this I had to change the logic to make it more loosely coupled and also stateless. Hence with more discussions with the team, I implemented a more stateless controller that uses the count of the messages not in flight from the input queue and also the number of instances active to balance the load. In the process I used Boto3 which is a Python library. The AWS API for Boto3 was easier to understand and simple to handle. The controller connects to SQS to monitor the number of messages and spawns EC2 instances with the image processor depending upon the load.

Implementation:
I implemented the Controller whose main function is to upscale the application depending on the number of messages that are input to the input SQS from the Web tier. The controller continuously polls the input SQS for any messages. Depending on the number of messages and also the number of currently active instances, it will create/start the necessary number of instances so that all the images can be processed by the instances that are active and the instances that have been spawned. The implementation is done using a Python program that uses Boto3 library which connects to the AWS API. The controller is started using a cron job and is placed in the same instance as the Web tier. It is placed in this tier in order to have more number of instances for processing the images. The controller continuously polls for messages if there are none in the queue. Once the message arrives, the controller will poll at an interval for the processing to happen and new messages to arrive. The controller has helper functions that are useful to start, create instances and also functions for the waiter, finding mathematical mode and getting the number of messages in the queue. Using these functions, the controller actively checks for the queue length and the current number of active instances and idle instances to spawn the necessary number of instances for processing. Finding the queue length is made more accurate with the help of finding the mathematical mode of 50 readings to make it more accurate.

Testing:
I initially worked on Unit testing for the controller. Where I tested in an iterative manner if each and every one of the conditions of the controller is working. Having completed the unit testing, I moved to do an end to end testing once the implementations of the other components were completed. On testing end to end, more number of failing edge cases arose and I tweaked the code so that the controller handled all those test cases as well. I worked along with my teammates in successfully testing vigorously for all the possible test cases in multiple permutations and in continuous iterations just to make sure that the code does not break or hang at any point in time. Upon completing these testing, we did a load testing for greater than 100 images to test for the speed of the entire execution.