CSE 546 - Project 1 Report

Sai Surya Kaushik Punyamurthula - 1220096111
Mrunalini Jitendra Dighe - 1219515557
Krisha Vijay Gala - 1222514124

1. Problem Statement

To build an elastic face recognition application using IAAS cloud resources from Amazon Web Services that can automatically scale in and out based on demand.
The cloud based application takes in multiple images as input and runs a face recognition algorithm on it to identify the underlying image. In this process, the core components of cloud computing are used which include auto scaling, rapid elasticity, measured services, broad range network access, and resource pooling. This problem is important as it addresses the most basic issue of auto scaling. This is finding an importance in the technology industry currently as it minimizes the cost based on the usability of the resources. In this project, explicit use of AWS EC2, SQS and S3 is made. The application is efficient in a way that it is based on AWS free tier which scales up and down automatically.

2. Design and Implementation

2.1 Architecture

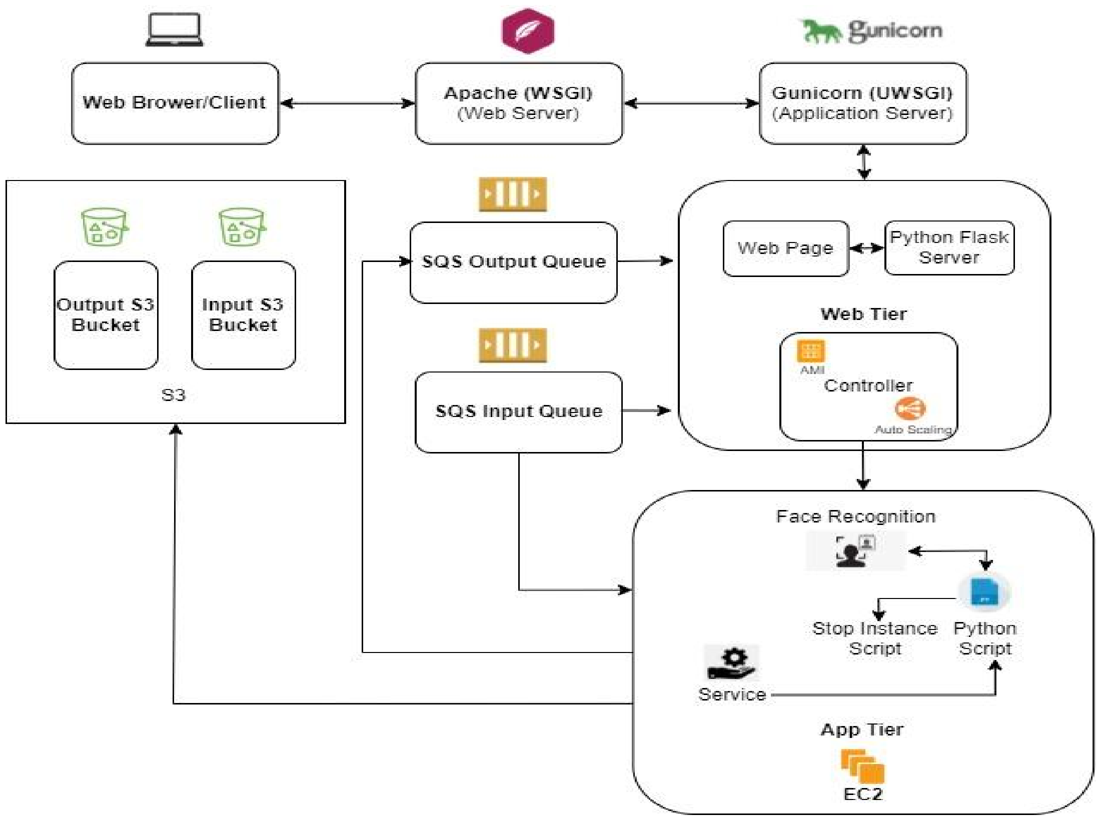


Fig. 1: Architecture Diagram

2.1.1 Architectural Summary:

As seen from Fig. 1, the application is basically divided into 3 components such as the web tier, controller and app tier. The controller is a part of the web tier itself. The controller and web tier are part of the same EC2 instance. A maximum of 19 instances can be launched , a logic that is handled by the controller. Hence, the controller is primarily responsible for autoscaling the instances based on the demand. When the web tier starts running, the controller and the app tier start running simultaneously at the same time. The user can access the user interface to upload images which is an html file using the public ip address. The app service is common in all app tiers and it is created using the controller. The controller takes the details of the image from the SQS input queue, processing it and placing the output in the SQS queue. The app tier helps place the message in the S3 bucket for persistence. The web tier takes the final result from the SQS queue and this is displayed to the user as output.

2.1.2 AWS Components:

1. **Elastic Cloud Compute - EC2**

   EC2 is a virtual server that enables users to deploy applications by creating instances. In this project, the web tier, controller and application services are hosted on EC2.

2. **Simple Queue Service - SQS**

   SQS is a queue service for messages that is deployed for communication between app tier and web tier. Input as well as output queue has been created using SQS.

3. **Simple Storage Service - S3**

   S3 is a storage service for objects which stores the input images and face recognition results in two separate S3 buckets.

4. **Amazon Machine Image - AMI**

   AMI provides information required to launch an EC2 instance. In this case, the AMI provided contains only a face recognition model. A custom AMI is created using this AMI which can receive images and then apply the face recognition model on it.

2.1.3 Implementation Components:

Web Tier:

1. The web tier consists of the web application and the controller which accommodates the logic for auto scaling the instances based on the requests received. The web application is responsible for web services which are based on Flask.
2. The web tier receives requests which contain the images to be processed. These images are then sent to the input queue which enables the scaling and acts as a buffer and message broker between the web tier and app tier.
3. The message sent to SQS includes the attributes ‘request_id’, ‘image_name’ and ‘image_data’.
4. The output poller keeps polling the response queue for messages and stores the request ID and corresponding face recognition result value in a class level dictionary structure.
5. As soon as the processing result is available in the dictionary the result is returned as response to the web client via application and web servers.

Controller:

1. It holds the main logic for auto scaling and is important in deciding when to scale up and when to scale down. The controller is a part of the E2 instance that has the web tier. The cron job scheduler is responsible for starting the controller inside of the EC2 instance.
2. It controls the number of instances in the app tier so that all the images can be run efficiently against the face recognition algorithm using AWS free tier which can use a maximum of 19 instances at a time.
3. The controller is made up of a Python3 SQS listener. It spawns new instances with the given AMI id and operations are performed based on states of other instances which are running.
4. The input SQS queue is continuously polled to check for input messages which have been placed by the web tier for processing.
5. Parallel processing of messages is done by the creation of additional instances by starting the idle instances, using the active ones or by creating new ones, while still maintaining the total count of 20 instances of AWS free tier.

The instances to be spawned or rebooted depends on the following criteria:

- When the queue length is 0, that is, there are no messages in the queue, the controller continues to pool the input SQS queue for new messages.
- When the queue length is greater than the maximum threshold then it looks for the idle instances and if there are any, they are started. If there are no idle instances, then new instances are created while maintaining a total of 19 app tier instances. If in case there are a few idle instances, then the idle instances are started and the deficit is fulfilled by creating new ones while maintaining the total of 19 instances.
- When queue length is lesser than the number of instances, the idle instances equal to the queue length are started. When the queue length is greater than the number of idle instances but less than the maximum threshold then first the need is tried to be addressed by starting all the idle resources. Then the deficit is taken care of by creating new resources while still maintaining the threshold of 19 instances.

Maximum instances which can be spawned under the threshold = Active instances + idle instances to be activated + newly created instances

App Tier:

1. The app tier is the processing system which processes the images and recognizes them based on the image classification in python.
2. The app tier instances are up scaled and down scaled based on the need and threshold of the instances.
3. The SQS listener in the app tier waits for the input queue to receive messages and stores the image data as an image file.
4. The face recognition program is then triggered with the path to the image file as an input attribute, which then recognizes the face in the image using available data points.
5. The results of face recognition are stored in the S3 output bucket and are sent to the output SQS queue, which are eventually received by the SQS listener of the web tier.
6. A python script to stop self shut down when called. This script is called after a certain time of application inactivity.

7. The inactivity of the application depends on two factors: if any images are being processed by the face recognition program and if there are any more messages in the queue which are yet to be processed by the app tier instances.

2.2 Autoscaling

Autoscaling is a feature of cloud computing and it ensures that the resources being used by the application are dynamically scaled up or down based on the demand.

2.2.1 Up-Scaling:

Controller enables the auto-scale up of the application. It keeps polling the input queue (SQS) consistently to check if there are any messages and to ensure that new messages added are processed. As soon as the input queue has a message, it will check the number of instances that are active and calculate how many instances need to be started (if instances are idle/stopped) or created in addition to the already active instances given that the threshold limit is not crossed. If there are idle instances, then the controller will first start those instances and only create new instances if all the idle instances have been started and there are still messages remaining to be processed in the input queue, provided threshold limit (20) is not exceeded. This way, by starting idle instances or creating new instances as per the demand, that is, the number of messages in the input queue, ensures auto scale-up of the application. If the threshold is not reached and there are messages in the queue, instances will be started/created to ensure maximum usage. If the length of the input queue is less than the number of idle instances, the necessary number of instances will be started. If the length of the input queue is more than the idle instances, these instances will be started and more instances will be created (total less than 20).

2.2.2 Down-Scaling:

Down-scaling of application takes place in the instances itself. If the input SQS queue is empty and no messages need to be processed, the length of the queue becomes 0. A script is made to run on each app instance which checks the length of the queue and the process running. If an instance is idle for more than 10 seconds and the queue length is 0, then the instance is automatically stopped. This ensures auto down-scaling of the application which is handled by each app instance. It is done on demand, that is, if there are no more messages in the input queue. This is how it is automatically scaled down.

3. Testing and Evaluation

3.1 Testing of the Individual Components:

3.1.1 Web Tier:

We tried to test the multiple aspects of the web tier by considering the following points thoroughly :

1. Tried uploading multiple images to check whether the functionality is okay or not.
2. Not letting the user proceed without uploading an image.
3. Retrieval of messages from the SQS queue.
4. Checking whether the image is getting sent to the SQS queue with the image attributes or not.
5. Storage in the S3 bucket.
6. Verification of results we obtain with the ones in the output of the S3 bucket.

3.1.2 Controller:

The controller was tested for the following points :

1. Creation of App Tier Instances.

2. Checking the state of the instances which are spawned.
3. Polling of the messages from the SQS queue.
4. Noting the queue depth and implementing the instance creation logic accordingly.
5. Check how many resources are started/created/spawned depending upon the number of messages in the queue and number of active instances.

3.1.3 App Tier:

1. Checking if messages are properly received from the SQS queue and there is no data loss.
2. Check if messages are getting sent to the output SQS queue with the output results.
3. Verification of the results of image classification.
4. Checking if the results obtained after image classification are getting uploaded to the S3 bucket properly.

3.2 Integration Testing:

Integration involves combining the individual components and then jointly performing testing so that the end to end flow is executed efficiently.

Step 1: We upload the images using the Web Tier and ensure that the image has all the necessary fields like image name and url. Then we also check if these fields were present as input in the SQS .

Step 2: We upload the images using the workload generator shared by the TAs to ensure the application works for concurrent requests and are accounted for.

Step 3: We check that the images are placed in S3 buckets.

Step 4: We let the controller poll messages continuously unlike in unit testing. The controller is tested for creating/starting all EC2 App Tier instances based on the load that is the number of messages.

Step 5: Check if the app tier instances get spawned, simultaneously check if the cron job which is a python program runs and consumes messages from the SQS queue to do image recognition. Later we check if the output is placed onto the SQS output queue and S3 output bucket.

Step 6: The final step includes to check the output we receive from the output SQS queue and the output S3 bucket and display it on the webpage.

3.3 Load testing:

A few example cases were tested as shown below to check if the desired result is generated

Length of Queue	**Processing Time**	**Number of Instances**	**Result**
10	239 sec	Active: 1, Idle: 0	As active instances were 1 (web tier) and we have a threshold of 19 instances, we make all the 19 instances active and shut them down after processing and inactivity is observed on the SQS queue.
50	149 sec	Active: 1, Idle: 19	With a threshold of 19 instances, we start the 19 instances which are in Idle state. After a period of inactivity the instances are shut down.

100	200 sec	Active: 11, Idle: 9	We have 10 active app tier instances and need additional 12, but as we have a threshold of 19 instances, we cannot exceed the total count of 19. So, all the idle 9 instances are started. Therefore a total 19 instances are active at a time and are shut down after processing and having no input on the input SQS queue.

The above test cases proved that the controller logic was efficient and scaled the resources accordingly.

Challenges:

1. Faced connection timeout but handled it by tweaking code and testing .
2. There were times when boot times were different due to AWS free tier limitations.
3. Faced a challenge with AWS credentials as the free trial expired and we created a new one.

4. Code and Setup

4.1 Web Tier:

webtier.py:

1. Comprises Web Tier components.
2. Uses Flask framework to enable communication with SQS and S3and build web interface.
3. Takes input files which the user uploads.
4. Puts messages with image attributes in the input queue (SQS).
5. Gets the classification result after the processing is complete from the output queue (SQS).
6. Displays this in the user interface/returns the response to the web client.

upload_form.html

1. This file has the script for the user interface containing all styling for the web page.
2. The page for the file upload is built using this script.
3. The user interface view has buttons for choosing the files to be uploaded, uploading the files and displaying the results.

gunicorn_config.py

1. This file has the gunicorn configuration specified which will be used by the gunicorn application server to cater according to the needs

controller.py

1. This file creates new instances or starts the idle instances.
2. It is an SQS listener to the input queue and it either creates instances using the AMI or starts idle instances to enable processing.
3. Auto up-scaling of the application is performed here.
4. It is responsible for terminating idle instances after being inactive (receiving no requests) for a while

configuration.properties

1. This file contains all the properties of AWS resources including the S3 bucket information, SQS queues information and the AMI details and the region details.
2. This file is accessed by the webtier.py and controller.py scripts to connect to the respective AWS resources.
3. The same file is used as part of the App Tier instances too, with the same usage as in the Web Tier instance.

miscellaneous

1. We additionally used systemd services to start the apache web server and gunicorn application server, which are done through apache configuration (responsible for tunneling of requests from external sources to the gunicorn server) and gunicorn service (responsible for executing the gunicorn application server with the gunicorn_config script file discussed earlier).

4.2 Application Tier:

apptier.py

1. This file processes the input images.
2. It is an SQS listener to the input queue and it initiates the processing of the image files as the messages are received from the queue.
3. This program writes the image data received as part of the message to a file and uploads the image to the input S3 bucket using the image name when a message is pushed onto the queue.
4. Then classification of the image is performed and the resultant label is pushed to the output queue which is received by the web tier.
5. Along with this, it is also stored in the S3 output bucket.
6. The message from the input queue is deleted upon the completion of the processing.
7. In the case of any processing failure, the message is sent back to the input queue and the processing is performed on it again.

stopInstance.py:

1. This file manages the auto-down-scaling of the application.
2. This script is configured to run on every instance to monitor whether the processing is still on or not in intervals of 60 seconds.
3. After all the messages from the input queue have been processed, this script checks if any processing is done for 60 seconds. If there are no processes running including the face_recognition script and the input queue has no messages to be processed, the current instance is instructed to initiate a self shut down.

miscellaneous

1. We additionally designed systemd services to start the apptier python script at the time of boot. As opposed to user data which is run during the creation of instances, services can be used while rebooting too.

4.3 Installation and Running:

Install the following packages into an EC2 instance:

1. Install python if required with version 3 and above.
2. Install pip package manager if required.

3. Install flask using pip for creating a web application using python.
4. Install paramiko using pip.
5. Install boto3 using pip for communicating with ec2 resources.
6. Install AWS CLI in the EC2 instance to configure aws credentials.
7. Install gunicorn using pip to host the flask application on an application server.
8. Install apache2 and libapache2-mod-wsgi packages to tunnel the requests from external IPs to gunicorn application server.

4.4 Steps to Setup Web Tier Instances:

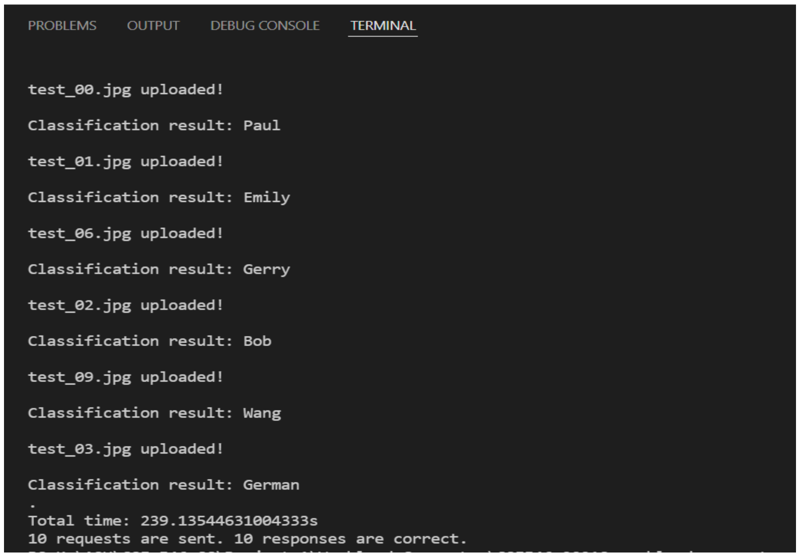
1. Place the '*webtier.py'* in /var/www/html/webtier (root directory of webtier) in the web-tier instance. This file is responsible for the web services provisioned and to render the user interface using the template '*upload_form.html'* html file which is to be placed in a subdirectory called *'templates'* in the current web tier directory.
2. Place the controller.py in the root directory of webtier. This file is responsible for scaling of app tier instances. A copy of the key pair pem file has to be placed in the same directory.
3. We can run the webtier flask application by starting the gunicorn service first and then the apache service using the command *'systemctl restart gunicorn.service'* or *'systemctl restart apache2'*. The controller can be started using the python3 run command - python3 controller.py & (to run in background).
4. We have a logging setup in place which logs the information from both the webtier flask application and controller python script to the 'webtier.log' file in the same directory, which can be monitored using cat/tail linux commands.
5. We have created a custom AMI for the app tier instances by creating an instance from the image shared in the project requirement document and placing additional logic on top of it which has been discussed earlier.
6. The controller spawns new instances or respawns idle instances based on requirement using the custom AMI.
7. The webtier application can be accessed from the web clients using the public URL: http://34.230.88.115/webtier/face_recognition for the rendered User Interface page and http://34.230.88.115/webtier/face_recognition/upload for uploading the images using workload generator.
8. In the case that a web browser is the client, after choosing the files, the controller will start picking up the images after clicking the Upload button. After the entire processing is done and the results are seen in the output bucket and the results can be viewed on the redirected UI page.
9. In the case that some other application (like workload generator) is the client, the images are directly sent to the upload API as requests and the corresponding results can be viewed in the client.

4.5 Steps to Setup App Tier Instances:

1. Confirm that the face recognition python scripts and related files are present in the ec2-user home directory of the instance launched using the AMI shared in the project requirement document.
2. Place the *'apptier.py'* and the *'stopInstance.py'* in the user home directory. Create a service file in the systemd directory to start the app tier application and configure it to start on boot and enable the service.
3. Create an AMI from the current app tier instance, which will be used by the controller to spawn instances on the go when required.

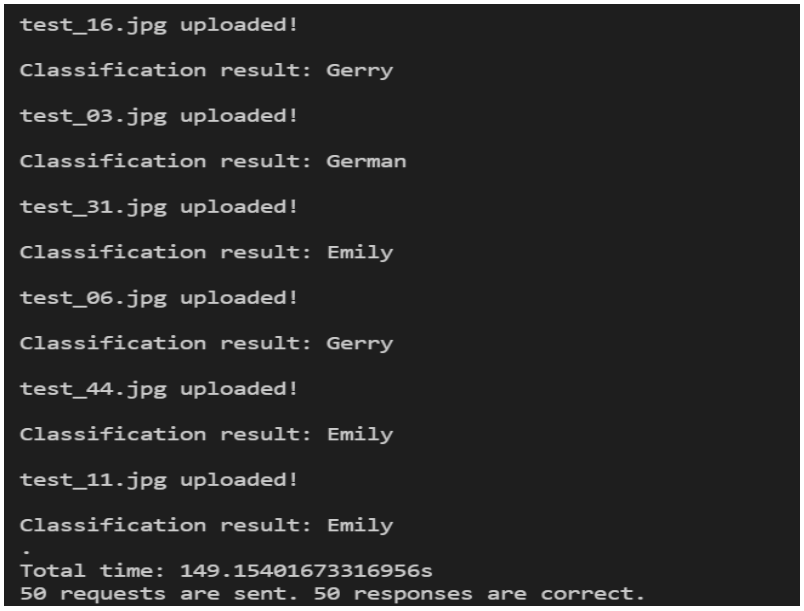
4. We have a logging setup in place which logs the information from both the apptier python application and stopInstance python script to the apptier.log' file in the home directory, which can be monitored using cat/tail linux commands.

5. Results



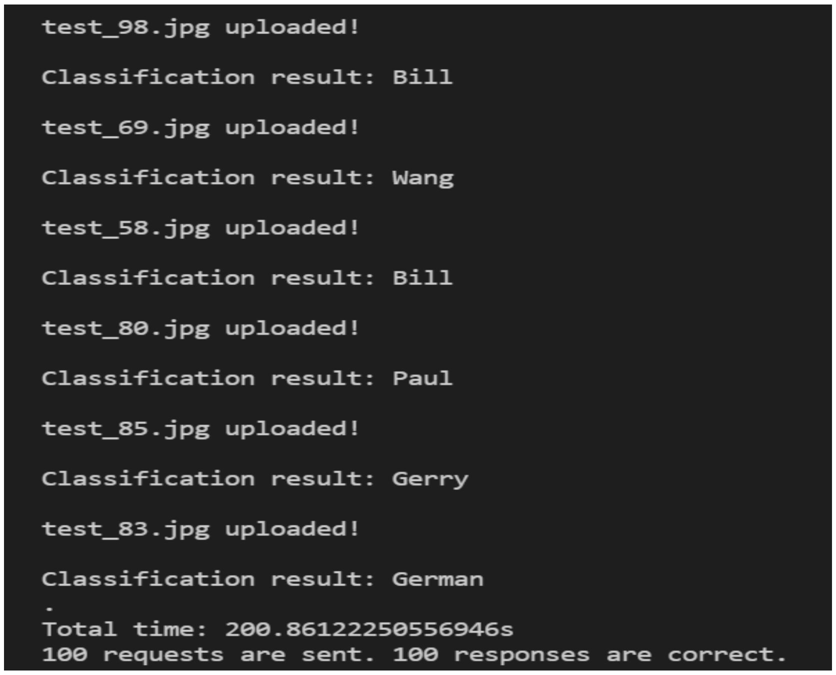
```
test_00.jpg uploaded!

Classification result: Paul

test_01.jpg uploaded!

Classification result: Emily

test_06.jpg uploaded!

Classification result: Gerry

test_02.jpg uploaded!

Classification result: Bob

test_09.jpg uploaded!

Classification result: Wang

test_03.jpg uploaded!

Classification result: German
.
Total time: 239.13544631004333s
10 requests are sent. 10 responses are correct.
```

10 Concurrent Requests with Multi-threaded Workload Generator



```
test_16.jpg uploaded!

Classification result: Gerry

test_03.jpg uploaded!

Classification result: German

test_31.jpg uploaded!

Classification result: Emily

test_06.jpg uploaded!

Classification result: Gerry

test_44.jpg uploaded!

Classification result: Emily

test_11.jpg uploaded!

Classification result: Emily
.
Total time: 149.15401673316956s
50 requests are sent. 50 responses are correct.
```

50 Concurrent Requests with Multi-threaded Workload Generator



```
test_98.jpg uploaded!

Classification result: Bill

test_69.jpg uploaded!

Classification result: Wang

test_58.jpg uploaded!

Classification result: Bill

test_80.jpg uploaded!

Classification result: Paul

test_85.jpg uploaded!

Classification result: Gerry

test_83.jpg uploaded!

Classification result: German
.
Total time: 200.86122250556946s
100 requests are sent. 100 responses are correct.
```

100 Concurrent Requests with Multi-threaded Workload Generator

Individual Contribution

(Mrunalini Jitendra Dighe)

I was involved in the design of the project. As one of our main aims was to keep the cost as minimal as possible, it was decided to follow the amazon free tier architecture.

Design:

I worked on the development of the controller part which accounted for managing the load balancing and monitoring SQS and the load on EC2 instances. The spawning logic was also implemented accordingly. But with further interactions of design improvisation it was found that this design was tightly coupled and involved a single point of failure. So I changed this to make a more robust design having loosely coupled components which were stateless. I implemented a logic to scale up and down the number of active instances based on maintaining the threshold to maximum available ones in the AWS free tier. In the development process I used the Boto3 Python library. Working on AWS API helps to integrate Python applications, libraries, scripts and AWS services ranging from Amazon S3, Amazon EC2. The controller, depending upon the load and number of messages in the queue, continuously polls it and spaws EC2 instances with image processor.

Implementation:

I worked on implementing the controller which was responsible for upscaling and downscaling the number of EC2 instances.The upscaling is dependent on the load that is input to SQS queue from the Web Tier. The controller looks for the messages in the SQS by continuously pooling it. It has a logic which takes into account the number of currently active instances, number of messages and will accordingly create new instances or start the idle instances. It ensures all the images will get processed by using active instances which become available to use or creating new ones. This is achieved with the help of Python BOTO 3 library which uses AWS API. The controller is initiated using the cron scheduler. The controller will poll after an interval and wait for messages to get processed and new messages to arrive. The controller has logical functionalities which are used to create new, start instances. It also includes a wait function which helps in finding the number of messages in the queue. The controller, using this logic, checks for the active instances count, number of idle instances to spawn, queue length and the necessary instances for processing. The queue length is found using mode of readings to increase the accuracy.

Testing:

I initially started with running unit test cases, after successfully running those I completed the end to end test cases according to the test plan. There were some failing edges which were addressed by tweaking the code and logic of the controller. After trying multiple combinations we were assured that the code didn't break inadvertently. Finally we did load testing which was done using 100 images. The application was able to handle this.

Individual Contribution

(Krisha Vijay Gala)

I was involved in the design of the project. As one of our main aims was to keep the cost as minimal as possible, it was decided to follow the amazon free tier architecture.

Design:

I was involved in creating the end to end flow and making a robust design which was loosely coupled. It did not have a single point of failure. I worked in gathering the referential knowledge about AWS Python SDk, BOTO 3 python libraries. In the inception of the requirement gathering phase, I thoroughly made sure that the cost stays minimum and only free tier resources are getting used. I worked on developing the user interface and connecting the web tier and app tier with S3 and SQS. I decided on choosing the architecture for using standard queues for input and output messages.

Implementation:

The implementation of the web tier primarily was based on the flask framework for backend. Html, CSS and javascript was used to address the front end. The flask framework made it very light weight and fast. The user interface is able to take in one or multiple images as input. On the user interface the images are uploaded using the "upload" button which makes calls to flask request methods in the backend. These images are sent to the input queue, this is done using functionalities from BOTO 3 library. This library also helps us connect with the various AWS resources. The image along with the image attribute is sent as a part of the message in the SQS input queue. There is a prompt provided which lets the user know about successful122 image uploading. The images are processed in the app tier and results are rendered to the user in the user interface by fetching the messages from the output SQS queue.

Testing :

I worked on connecting the components to the web tier and then unit testing their functionality. I tested the image upload functionality using multiple images. The maximum number of images we used were 100. The functionality is tested using multiple images with minimum one image.The instances were also checked in the AWS console to see if they are running and active or not.

The logic of scaling up and down was also done successfully, with the instance count increasing and decreasing respectively. The test results from the output SQS queue were checked against the classification results provided by the machine learning model and they matched correctly.

End to end testing was also done to check the workflows. The load testing was done by testing the application with multiple messages.

Individual Contribution

(Sai Surya Kaushik Punyamurthula)

I was involved in the design of the project. As one of our main aims was to keep the cost as minimal as possible, it was decided to follow the amazon free tier architecture.

Design:

I worked on analyzing how to make the web application available to external devices. Finally decided on using Apache2 WSGI server as the web server to expose the end points to the internet. The requests are tunneled to the web application or the application server if any. I have come up with the idea to use the gunicorn application server to host the flask application. This enables processing multiple/concurrent requests using multi-threading. A data structure would be required to map the response to the correct request as multiple threads would be polling for the response. This can be achieved by having the data structure in this case a python dictionary as part of a class which can be called in any of the created threads.

Implementation:

The implementation of the gateways (web server and application server) requires installing the required modules in the instance. Once we install the apache2 mod_wsgi packages on the instance we have to configure apache to proxy the requests to apache server to the application server which is hosting the flask application, we do so by specifying the request url which has to proxied and the application local url and port. Gunicorn has to be configured to handle an extended timeout of the requests and multiple workers which run the concurrent requests separately based on the number of connections required. This enables us to deal with each response individually. Gunicorn gevent runs an event loop and monkey patches Python to use non-blocking IO which enables us to not worry about the management of threads and python to write synchronously using a concept called pseudo threads. In order to have the data persist in the data structure which maps the response to request we have to use class level variables which makes it possible to be accessed by different threads without having to break when concurrent updates are made to it.

Testing:

The application was initially tested with a standalone flask application but we were not able to run either concurrent requests or expose it to external devices. We then used Apache to expose the application, which worked quite well sequential multiple files upload requests. This approach failed when we made concurrent requests however. Upon careful evaluation and research I have come up with the idea to use gunicorn gevernt with worker classes to address this issue. We were successfully able to then make concurrent requests and send the right response to the request made, which was not possible earlier. We have done the testing by uploading various numbers of files to test how the application handles the load starting with a mere 5 concurrent requests with one image file each and then progressing to 10, 50 and 100 requests. The application was also tested hitting the url and also using the multi threaded workload generator . The final demo was done using the multi threaded workload generator and was successful as many iterations of end to end testing were performed. The test cases were so designed so that the instances being spawned and active held a count that was within the range of the AWS free tier.