CSE 546 — Project1 Report

Hemendu Roy
Radhika Ganapathy
Yihui ZENG

1. Problem statement

In this project, we aim to build an elastic application which can automatically scale out and in on demand and cost-effectively by using the IaaS cloud resources from Amazon Web Services (AWS). AWS is the most widely used IaaS provider and offers a variety of compute, storage, and message services. Our application will offer a meaningful cloud service to users, and the technologies and techniques that we learn will be useful for us to build many others in future.

In this project, we aim to build a scalable face recognition service on the cloud which has all the software components loosely coupled. Every component of AWS used is elastic and hence, the project can be scaled up and down as per the number of requests coming in.

· Our cloud application performs face recognition on user-provided images using the provided deep learning model, which then returns the recognition result as output to the users.

· Each image input through the website is a request to our application and our application can handle multiple requests concurrently.

· It automatically scales out when the request demand increases, and automatically scales in when the demand drops.

· The application should handle all the requests as fast as possible, and it should not miss any requests. In addition, the recognition requests should all be correct.

2. Design and implementation

2.1 Architecture

We have built a Flask application for the web tier so that the user can upload multiple images. The web tier always runs on an AWS instance which sends messages with image information (encoded image and image name) to the Request queue (FIFO SQS). App instances created, polls for a message from the request queue and processes the same to get the recognition results. Each of these results is then sent back to the web tier using a Response queue (FIFO SQS) and these results are then displayed in a tabular format on the web tier.

- The SQS is responsible for maintaining two communication channels between the web tier instance and application tier instances. The FIFO queues will result in each message being processed exactly once by the app instances so that there is no duplication of the recognition results.

- The controller is a part of the web tier and is responsible for scaling in and scaling out the number of instances based on the load of requests.
- An app tier instance is spawned by the controller based on the auto scaling parameters that have been set. Each such instance contains a face recognition model and a python script that perpetually polls the request queue for incoming requests. Once a request is detected, it begins processing it in a series of steps.
    - Consume a message from request queue
    - Use the provided face recognition model to get a classification result
    - Send the response to the response queue
    - Write the input and outputs to their respective S3 buckets
    - Delete the message from the request queue

  Now, the python script returns to its polling state and waits for the next request

The high-level architecture of the cloud-application is shown in Fig. 1.

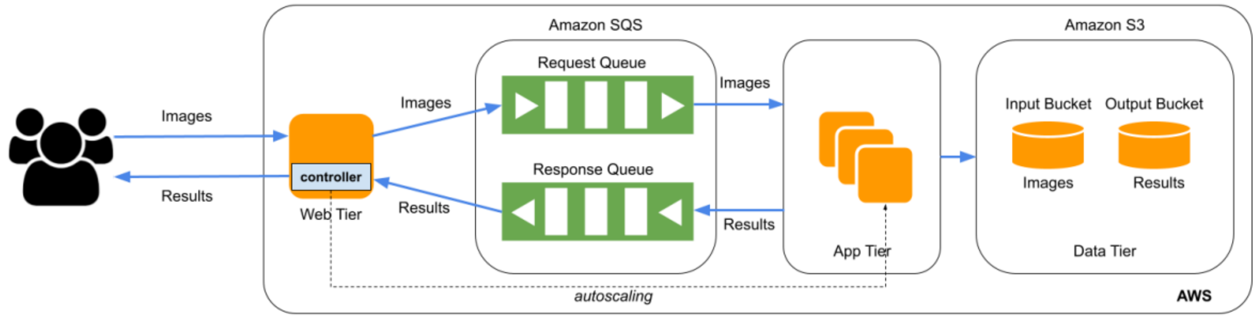


Fig. 1

2.2 Autoscaling

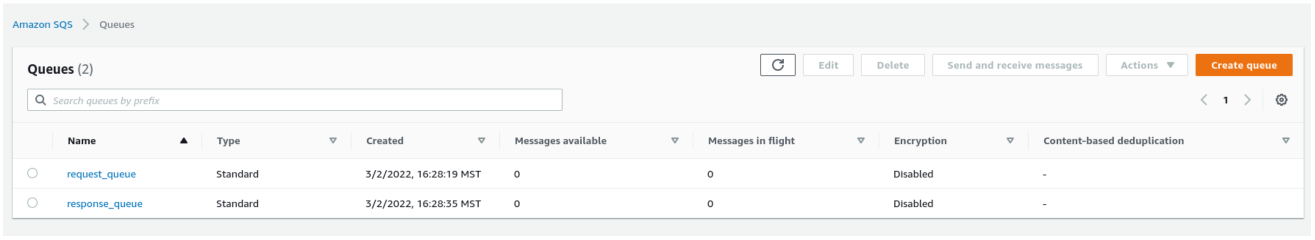
Autoscaling is one of the crucial reasons for hosting applications on the cloud. This project utilizes the Auto Scaling property of the cloud, so that the application resources are used efficiently. The web tier instance takes care of scaling up the instances, based on the number of requests made, considering one image is equal to a request. We will be using at most x app instances in this project and each instance processes the images to get the recognition result. If the number of requests is more than 100 at a particular time, the request-messages are held in the Request queue until there is an app instance available to pick up the request. The code embedded in each of the app instances will poll the SQS request queue, in order to check if there are messages in the Request queue. If there are no messages left in the Request queue, the app instance will stop itself automatically, so that the resources are not wasted and thus the application is scaled down by the app instance itself.

3. Testing and evaluation

Testing is an important part of building a cloud application, where you check if the application handles the requests concurrently and if it auto-scales, so that minimum resources are used to maximize the efficiency of the final product.

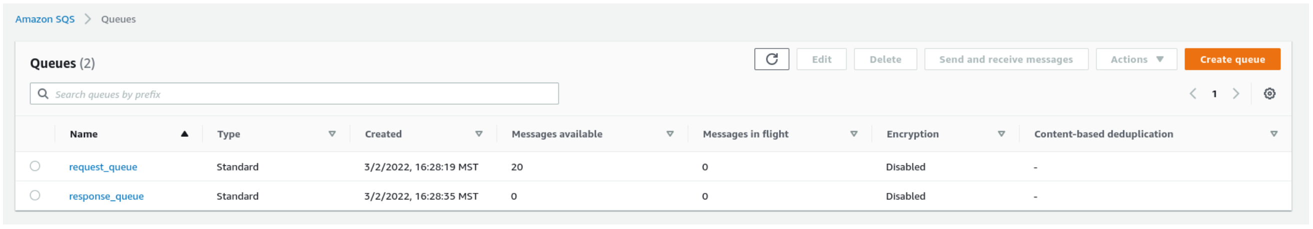
In this project, we tested our application for stress (work load) and also efficient auto-scaling of the EC2 instances. The images below portray that the application works as expected and the deployed product is efficient and fast.

A workload generator is used to replicate concurrent requests. Our system was tested on this and successfully executed the load of 100 requests in 130 seconds.



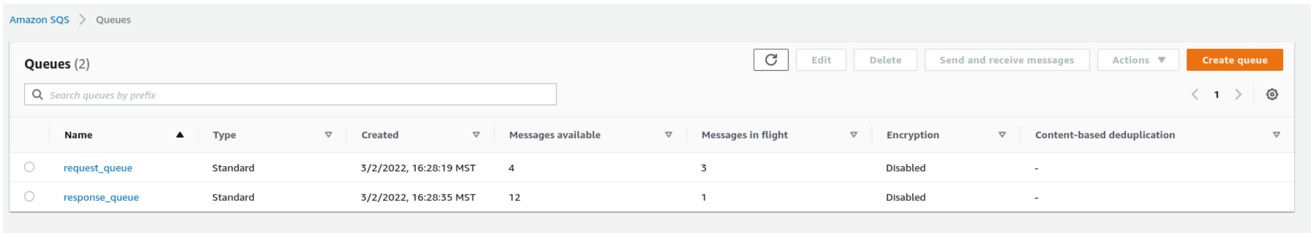
Name	Type	Created	Messages available	Messages in flight	Encryption	Content-based deduplication
request_queue	Standard	3/2/2022, 16:28:19 MST	0	0	Disabled	-
response_queue	Standard	3/2/2022, 16:28:35 MST	0	0	Disabled	-

Fig. 2 SQS set-up.



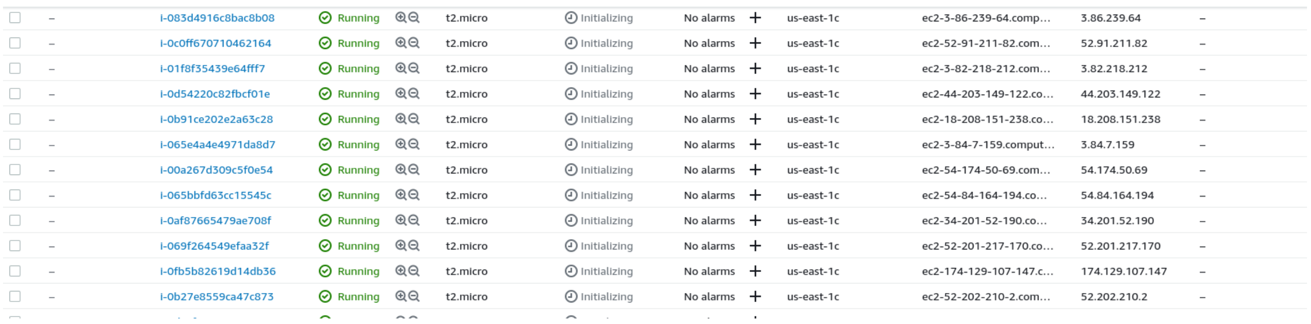
Name	Type	Created	Messages available	Messages in flight	Encryption	Content-based deduplication
request_queue	Standard	3/2/2022, 16:28:19 MST	20	0	Disabled	-
response_queue	Standard	3/2/2022, 16:28:35 MST	0	0	Disabled	-

Fig. 3 SQS messages sent



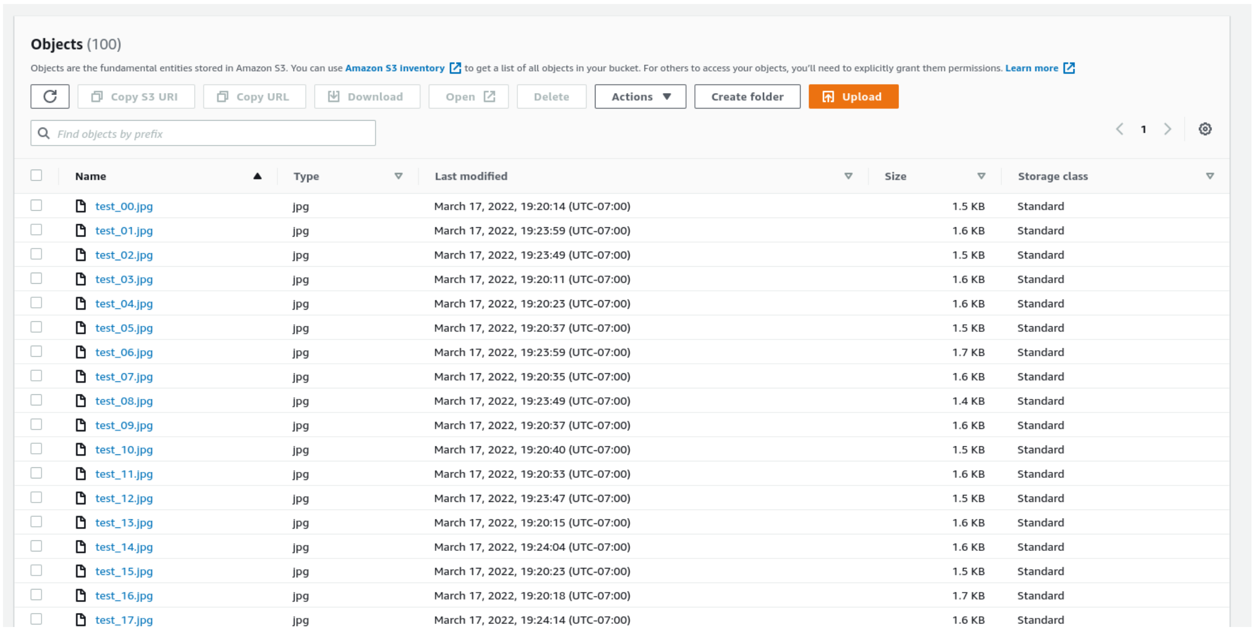
Name	Type	Created	Messages available	Messages in flight	Encryption	Content-based deduplication
request_queue	Standard	3/2/2022, 16:28:19 MST	4	3	Disabled	-
response_queue	Standard	3/2/2022, 16:28:35 MST	12	1	Disabled	-

Fig. 4 SQS messages processed.



☐	–	i-083d4916c8bac8b08	Running	t2.micro	Initializing	No alarms	+	us-east-1c	ec2-3-86-239-64.comp...	3.86.239.64	–
☐	–	i-0c0ff670710462164	Running	t2.micro	Initializing	No alarms	+	us-east-1c	ec2-52-91-211-82.com...	52.91.211.82	–
☐	–	i-01f8f35439e64fff7	Running	t2.micro	Initializing	No alarms	+	us-east-1c	ec2-3-82-218-212.com...	3.82.218.212	–
☐	–	i-0d54220c82fbcf01e	Running	t2.micro	Initializing	No alarms	+	us-east-1c	ec2-44-203-149-122.co...	44.203.149.122	–
☐	–	i-0b91ce202e2a63c28	Running	t2.micro	Initializing	No alarms	+	us-east-1c	ec2-18-208-151-238.co...	18.208.151.238	–
☐	–	i-065e4a4e4971da8d7	Running	t2.micro	Initializing	No alarms	+	us-east-1c	ec2-3-84-7-159.comput...	3.84.7.159	–
☐	–	i-00a267d309c5f0e54	Running	t2.micro	Initializing	No alarms	+	us-east-1c	ec2-54-174-50-69.com...	54.174.50.69	–
☐	–	i-065bbfd63cc15545c	Running	t2.micro	Initializing	No alarms	+	us-east-1c	ec2-54-84-164-194.co...	54.84.164.194	–
☐	–	i-0af87665479ae708f	Running	t2.micro	Initializing	No alarms	+	us-east-1c	ec2-34-201-52-190.co...	34.201.52.190	–
☐	–	i-069f264549efaa32f	Running	t2.micro	Initializing	No alarms	+	us-east-1c	ec2-52-201-217-170.co...	52.201.217.170	–
☐	–	i-0fb5b82619d14db36	Running	t2.micro	Initializing	No alarms	+	us-east-1c	ec2-174-129-107-147.c...	174.129.107.147	–
☐	–	i-0b27e8559ca47c873	Running	t2.micro	Initializing	No alarms	+	us-east-1c	ec2-52-202-210-2.com...	52.202.210.2	–

Fig. 5 App tier instances running.

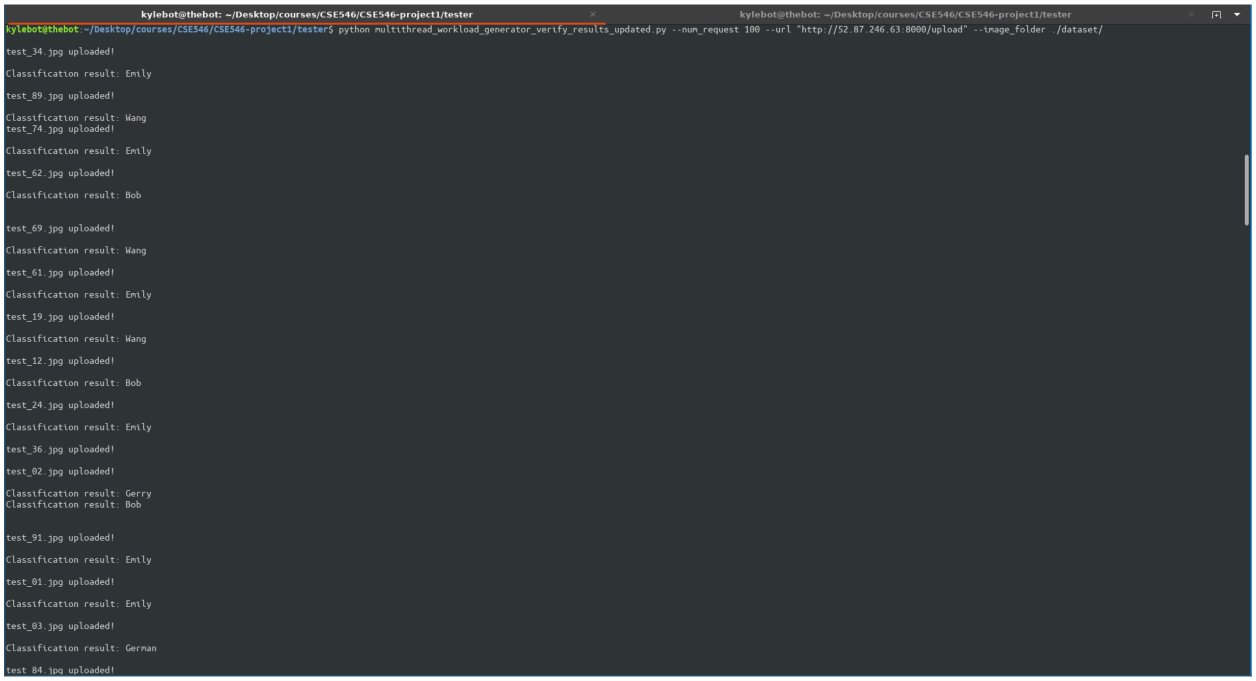


Objects (100)

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_00.jpg	jpg	March 17, 2022, 19:20:14 (UTC-07:00)	1.5 KB	Standard
☐	test_01.jpg	jpg	March 17, 2022, 19:23:59 (UTC-07:00)	1.6 KB	Standard
☐	test_02.jpg	jpg	March 17, 2022, 19:23:49 (UTC-07:00)	1.5 KB	Standard
☐	test_03.jpg	jpg	March 17, 2022, 19:20:11 (UTC-07:00)	1.6 KB	Standard
☐	test_04.jpg	jpg	March 17, 2022, 19:20:23 (UTC-07:00)	1.6 KB	Standard
☐	test_05.jpg	jpg	March 17, 2022, 19:20:37 (UTC-07:00)	1.5 KB	Standard
☐	test_06.jpg	jpg	March 17, 2022, 19:23:59 (UTC-07:00)	1.7 KB	Standard
☐	test_07.jpg	jpg	March 17, 2022, 19:20:35 (UTC-07:00)	1.6 KB	Standard
☐	test_08.jpg	jpg	March 17, 2022, 19:23:49 (UTC-07:00)	1.4 KB	Standard
☐	test_09.jpg	jpg	March 17, 2022, 19:20:37 (UTC-07:00)	1.6 KB	Standard
☐	test_10.jpg	jpg	March 17, 2022, 19:20:40 (UTC-07:00)	1.5 KB	Standard
☐	test_11.jpg	jpg	March 17, 2022, 19:20:33 (UTC-07:00)	1.6 KB	Standard
☐	test_12.jpg	jpg	March 17, 2022, 19:23:47 (UTC-07:00)	1.5 KB	Standard
☐	test_13.jpg	jpg	March 17, 2022, 19:20:15 (UTC-07:00)	1.6 KB	Standard
☐	test_14.jpg	jpg	March 17, 2022, 19:24:04 (UTC-07:00)	1.6 KB	Standard
☐	test_15.jpg	jpg	March 17, 2022, 19:20:23 (UTC-07:00)	1.5 KB	Standard
☐	test_16.jpg	jpg	March 17, 2022, 19:20:18 (UTC-07:00)	1.7 KB	Standard
☐	test_17.jpg	jpg	March 17, 2022, 19:24:14 (UTC-07:00)	1.6 KB	Standard

Fig. 6 S3 bucket storing input and output information.



kylebot@thebot: ~/Desktop/courses/CSE546/CSE546-project1/tester
kylebot@thebot: ~/Desktop/courses/CSE546/CSE546-project1/tester

```
kylebot@thebot:~/Desktop/courses/CSE546/CSE546-project1/tester$ python multithread_workload_generator_verify_results_updated.py --num_request 100 --url "http://52.87.246.63:8000/upload" --image_folder ./dataset/

test_34.jpg uploaded!

Classification result: Emily

test_89.jpg uploaded!

Classification result: Wang
test_74.jpg uploaded!

Classification result: Emily

test_62.jpg uploaded!

Classification result: Bob


test_69.jpg uploaded!

Classification result: Wang

test_61.jpg uploaded!

Classification result: Emily

test_19.jpg uploaded!

Classification result: Wang

test_12.jpg uploaded!

Classification result: Bob

test_24.jpg uploaded!

Classification result: Emily

test_36.jpg uploaded!

test_02.jpg uploaded!

Classification result: Gerry
Classification result: Bob


test_91.jpg uploaded!

Classification result: Emily

test_01.jpg uploaded!

Classification result: Emily

test_03.jpg uploaded!

Classification result: German

test_84.jpg uploaded!
```

Fig. 7 Output terminal displaying results.

4. Code

The project uses python programs that implement the different abstractions of the boto3 services that have been used in the Flask app, which would run in the 'app.py' file. All the three services are independent of each other and from the main module. They are loosely coupled such that they could be reused in other projects too with minimal or no changes.

The three python modules using AWS SDKs in this project are:-

- Amazon Simple Queue Service – It is a distributed message queuing service. We use methods that are required to perform basic operations on SQS Service like creating, deleting, sending, and receiving messages using 'sqs' resource of boto3 SDKs in Python.
- Amazon Simple Storage Service - It is a service offered by Amazon Web Services that provides object storage through a web service interface. The boto3 SDKs also provide services to handle S3 buckets and their methods help in uploading and downloading files to and from the S3 buckets.
- Amazon Elastic Compute Cloud - It allows users to rent virtual computers on which they can run their own computer applications. The boto3 SDKs contain methods that are required to perform basic operations on EC2, such as starting or terminating (stopping) an instance.

4.1 Web Tier

The web tier runs on a Flask application which is launched using the main.py script. In this application users can upload an image. This image is then encoded into a base64 format and passed as a message into the request queue along with the label of the image which serves as an identifier. Apart from this, the web tier keeps polling the response queue to identify the recognition results of the images that were pushed into the request queue and once it matches the result using the identifier, the results are displayed.

4.2 App Tier

1. Functions
    - *read_one_request()* - It reads a message from the SQS Request Queue

      Input - None

      Returns - fname, content, msg

      fname - This is the filename of the image sent in the request message. It is retrieved from the ‘imageName’ attribute of the message.

      content - This is the base64 decoded message i.e. the actual image.

      msg - This is the message metadata

    - *recognize()* - It uses the deep learning model to get a prediction result

      Input - content

      content - This is the base64 decoded message i.e. the actual image.

      Returns - label

      label - This is the prediction result from the deep learning model. If no result is obtained, ‘None’ is returned instead.

    - *send_one_response()* - It sends a message to the SQS Response Queue

      Input - fname, label

      fname - This is the filename of the image sent in the request message. It is retrieved from the ‘imageName’ attribute of the message.

label - This is the prediction result from the deep learning model.

Returns - None

- *delete_message()* - deletes a message from the SQS Request Queue using the ‘ReceiptHandle’ attribute.

  Input - msg

  msg - This is the message metadata

  Returns - None

It consumes a message from the Request Queue, uses the deep learning model to get a prediction result and sends the result to the SQS response queue. Processed messages are deleted from the Request Queue. Additionally, the input and output data is stored in separate S3 Buckets.

Steps to run our project :-

- to run the web_tier: *cd web_tier; gunicorn --bind 0.0.0.0 -w 20 -t 1800 wsgi:app*
- to run the controller : *cd controller; python3 watchdog.py*
- we created a AMI for app-tier
- controller will launch instances from the created AMI

5. Individual Contributions

5.1 Hemendu Roy

I primarily worked on designing the App Tier Instances. This entailed writing Python code as described in *Section 4.2*.

The script performs the following tasks,

- Consume a message from request queue
- Use the provided face recognition model to get a classification result
- Send the response to the response queue
- Write the input and outputs to their respective S3 buckets
- Delete the message from the request queue
- Continue polling the request queue for new messages

Once the app tier scripts were ready, I configured them to be run automatically upon instance startup. At first, I configured this using *cron*. Later, this was converted to a *systemd* service. Lastly, I created an AMI to be used by the controller to scale in/out the app tier instances.This AMI was subsequently modified during the integration and testing phases.

5.2 Radhika Ganapathy

I mainly worked on the web tier interface and built a flask application to implement the same.

- The web app has an interface that enables users to upload an image to the web application.
- The image uploaded is then encoded into base64 format.
- Created a Request Queue using Amazon's SQS Service.
- Every time an image is uploaded, a message containing the encoded image along with the label is sent to the queue.
- Web app keeps polling the response queue to see if the result is processed and ready to be displayed.
- Once the result is matched using an identifier, the result is displayed.