

CSE 546 — Project Report

Yash Bhokare, Shivangi Singh, Sayali Papat

1. Problem statement

The aim of this project is to build an elastic application that can automatically scale out and scale in on demand. The application provides an image Classification service to users. Images uploaded by the users are classified and the corresponding results along with the input image are stored in Amazon S3 buckets.

2. Design and implementation

2.1 Architecture

A high-level description of the image-classifier model

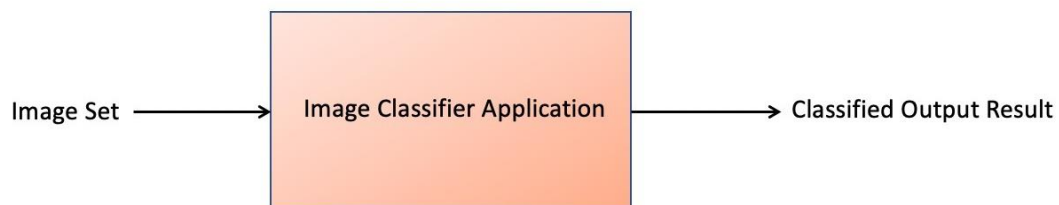


Figure 1

In Figure 1, At first, the user selects and inputs multiple images to the image classifier application. The image classifier app runs a classifier and classifies all the images and sends the output back to the user.

A detailed description of the image-classifier model for Figure 2:

1. Web-Tier: On web-tier start, the user selects multiple images on the web-tier and uploads them. The web-tier processes and sends the request to the S3 'input' bucket, where the images are stored, and a URL is generated for the location of the stored image. The URL is returned to the web-tier. The web-tier then passes the input-app to the SQS Launch Queue.

2. Launch Phase: The Launch Queue is used to maintain all the input image URLs uploaded by the user. The Launch controller is used to scale up the instances based on the load. The launch controller checks the launch queue and creates EC2 instances of the image-classifier. Here the launch controller also keeps a check on the status of EC2 instances and maintains a count of 'running' and 'free' instances of image-classifier. It creates a maximum of 20 instances when the load is too high.

3. EC2- Instance of image-classifier: The EC2 instance for image-classier when started fetches input-URL from Launch Queue and deletes the message from the queue. As it has fetched the input it passes the message to Monitor Queue stating that it started processing the image. It then fetches the image from S3 'input' bucket based on the given input-URL. The image is classified, and the output result is stored on the S3 bucket and then sent to the Output Queue. The instance then again checks for any new messages inside the Launch Queue and then repeats the process. If there are no more input-URLs to process it terminates itself which helps in scaling down the application as there's not much load.

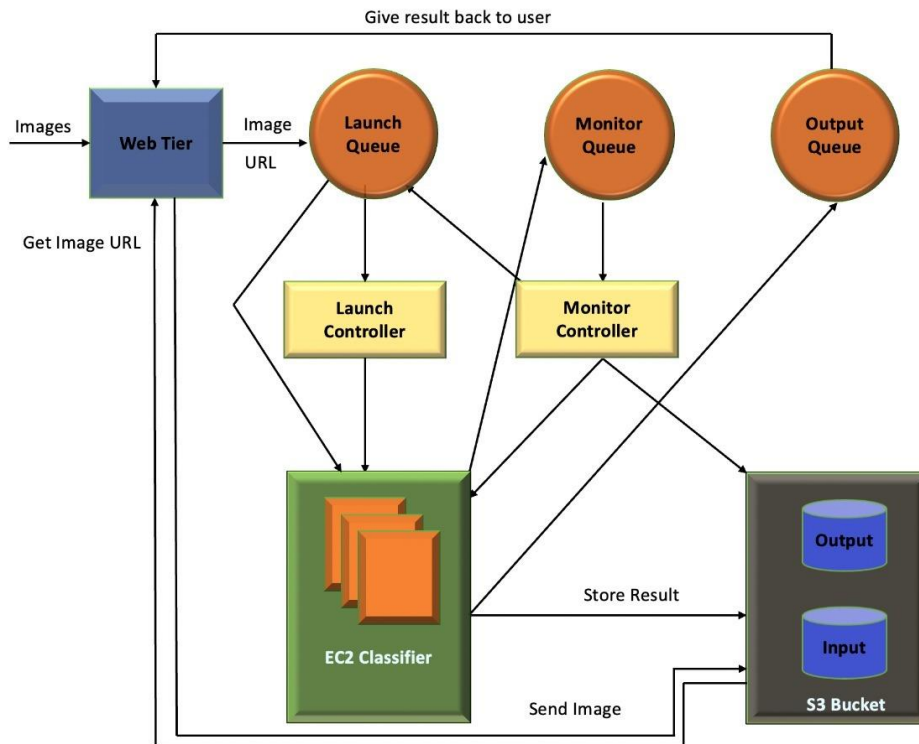


Figure 2

4. Monitor Phase: The monitor controller is used for handling multiple error scenarios. It monitors the health of instances for a given input and terminates them if it can't handle the load. It monitors the Monitor Queue and fetches the image and instance id from it. It then checks for output on the S3 bucket for the given input after some duration. If there's no Output for the given Input it terminates the instance and sends the message back to the Launch Queue. For a given image this would happen for a maximum of 3 tries after which it sends a message to Output Queue which states an error for giving an image.

5. Output Phase: The web-tier fetches the output result data from the Output Queue and displays it to the user.

2.1.1 Workflow:

The user selects multiple images on the web-tier and then uploads them. The web-tier makes a call to the S3 bucket and stores the images on the 'input image' folder and it returns back a URL for each image. Now the web-tier uploads all the images with params(inputURL, inputString, retryCount: 0) to the Launch Queue to process further.

The App-Controller is hosted on an EC2 instance and it contains the Launch and Monitor controller which are always running in parallel. The launch controller keeps a check for messages from the Launch and Monitor Queue. If there are any messages inside the Launch queue it is checked against the number of instances already running and it fetches a total count of free instances. Now the launch controller creates new instances based on the given demand of requests and assigns a tag to each EC2 instance. After processing the number of messages in Launch Queue the controller is given a time-out of 1 min to wait to check the next SQS requests.

Now the Ec-2 instance is created and is at first in the pending state. Once it's in the running state a bash script is executed which was added inside the user-data while creating that instance. This bash

script runs a python image-classifier. This code first checks for the message from the Launch Queue if there's any message it processes, stores locally, and deletes it. Now the message is sent to the Monitor Queue with Status updated as Running and Retry count increased by 1 (Retry Count is the total count for the number of times a specific image was tried to classify on an instance). Now the input image is fetched from the S3 bucket using the authentication credentials and provided to the image-classifier model. The model processes the image and gives an output in string format. The output along with the input image name is stored on the S3 bucket. Also, the image result is sent to the Output Queue along with Status completed. Now the code tries to check if there's any new message inside the Launch Queue and process it if any. If there's nothing inside the Launch Queue, it terminates itself and scales down the application resource when the demand is reduced.

The Monitor controller is used to check the health of the instances for the given image. It keeps track of Monitor Queue and keeps on checking for new messages. Once it receives a message it adds a timer to it and deletes the message from the queue. Now the timer is added to check the output bucket after the given time and if the output exists it means the images were classified successfully. If due to some reason there's no Output for a given image it terminates the instance and sends the message back to the Launch Queue to process it again. In this, if the number retry counts reach 3 it passes the message to Output Queue stating an error in processing this image.

The Web-tier displays the output as soon as the messages are received in the Output Queue. In this, there's a spinner added to each image which flips like a coin when output for the image is received. So when images flip we can see the output for that image. It has a polling time of 20 secs on which it keeps on checking the output queue and deletes the message once received. The web-tier would display pop-up stating images processed successfully after all outputs are received. It then stops the message check on the output queue.

2.2 Autoscaling:

Scaling in and scaling out is done at the application tier. Depending on the current load we perform the autoscaling. We ensure autoscaling with following components:

SQS: SQS decouples the components of our application, and queues the requests received by the web tier. It keeps the request in the queue till it is processed by EC2 instances. We have an upper limit of 20 on the number of instances in running state at any point of time, thus SQS queues all the pending requests when it reaches this threshold. We are using the length of the queue as the metric to determine the current load and perform scaling in and scaling out of compute resources. We check the length of the queue, the current number of running instances which are processing images, and then decide how many new instances are to be spawned. In order to avoid making continuous requests to the SQS queue we are using long polling, which reduces the cost of using SQS by eliminating the number of empty responses.

Launch Controller: Scaling out of EC2 instances is done at application controller level. The launch module of the controller uses the length of the launch queue as a metric to scale up the instances, making sure that at any point in time the instances count does not go beyond 20. The logic we have implemented is to process the images as soon as they are received to provide fast service to end users. To achieve this, we try to match the number of messages visible in the launch queue and the number of running instances. If the number of SQS messages is higher than 20, the launch controller will spin up EC2 instances making sure that their count does not go beyond 20. This ensures the scaling up of the app tier depending on load. The deployed app instances will process one message each from the queue parallelly, while the remaining messages will be in the queue and will be processed once the

app instance is ready to process the next request. With an implementing queue, we make sure that all the requests of the user are processed, and no request is lost.

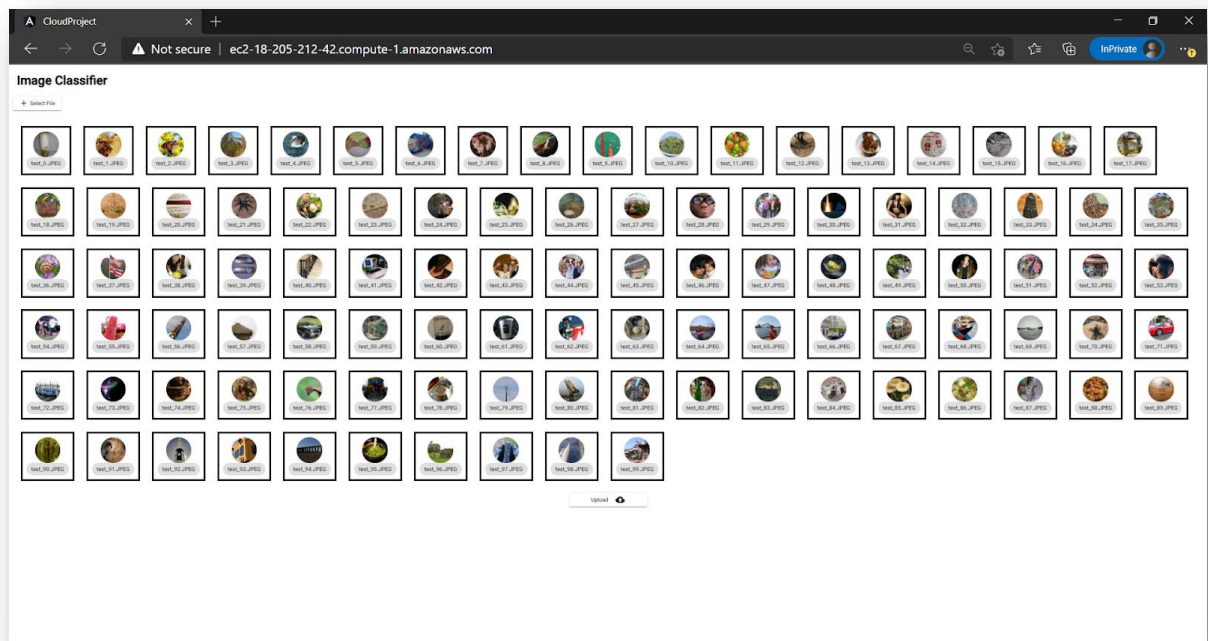
App Instance: Scaling in is done at application instance level. It processes the current image, sends its output to S3 and output queue. After completing the current request, it checks for the next message in the queue and performs the automatic scaling down operation if there is no message to be processed in the launch queue, by terminating itself.

3. Testing and Evaluation:

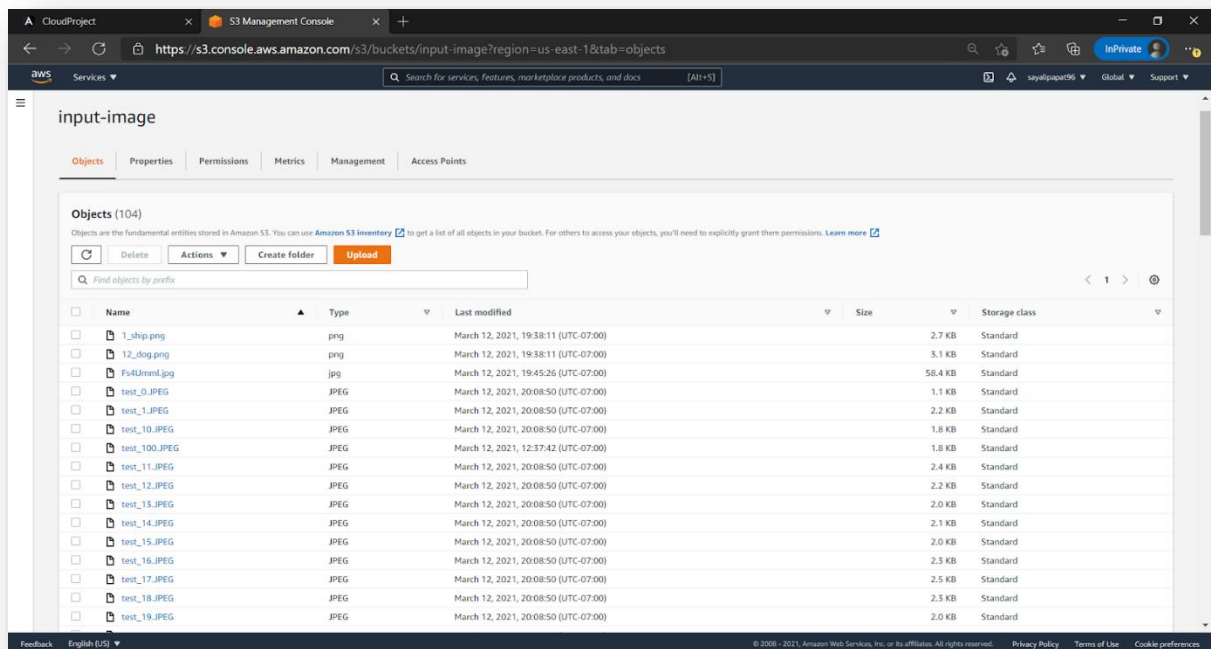
Performed unit testing and integrated testing on application iteratively to improve the performance and response time of application. We tested following use cases:

1. Tested with 100 images: We uploaded 100 images and checked the autoscaling of the EC2 instances. On receiving the output, we compared it with the classification results provided by the TA to check the accuracy of the results. With this test we also verified, the check where requests are queued in the launch queue when the total instances are running at their threshold number (20) and processing the images. We checked that all requests of a user are processed, and none is lost.

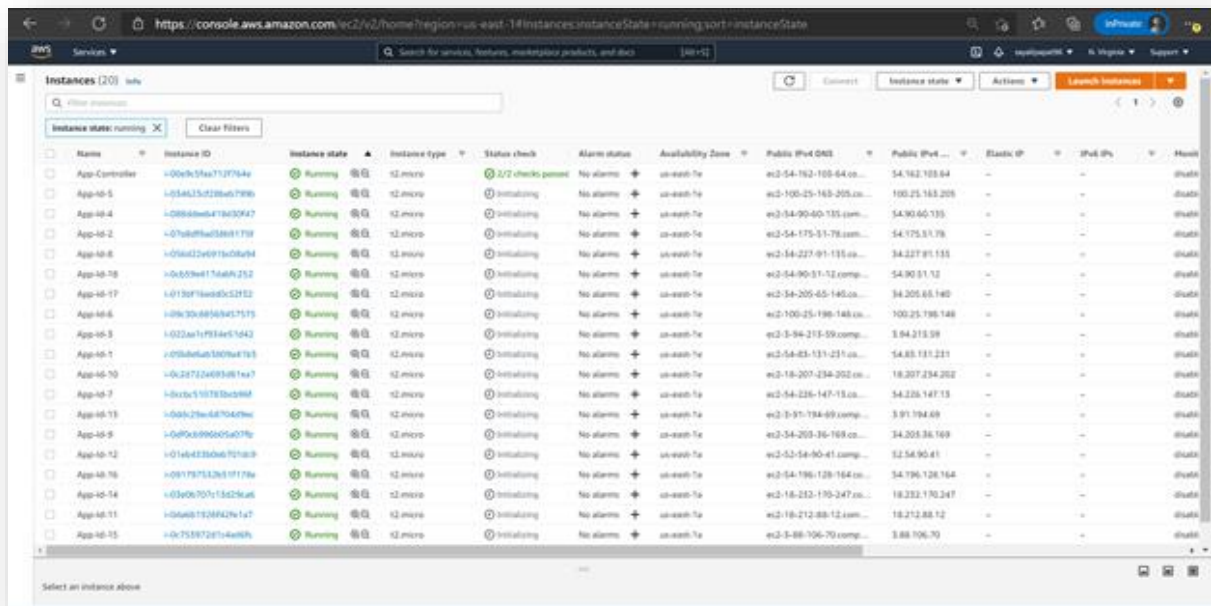
100 images to be uploaded:



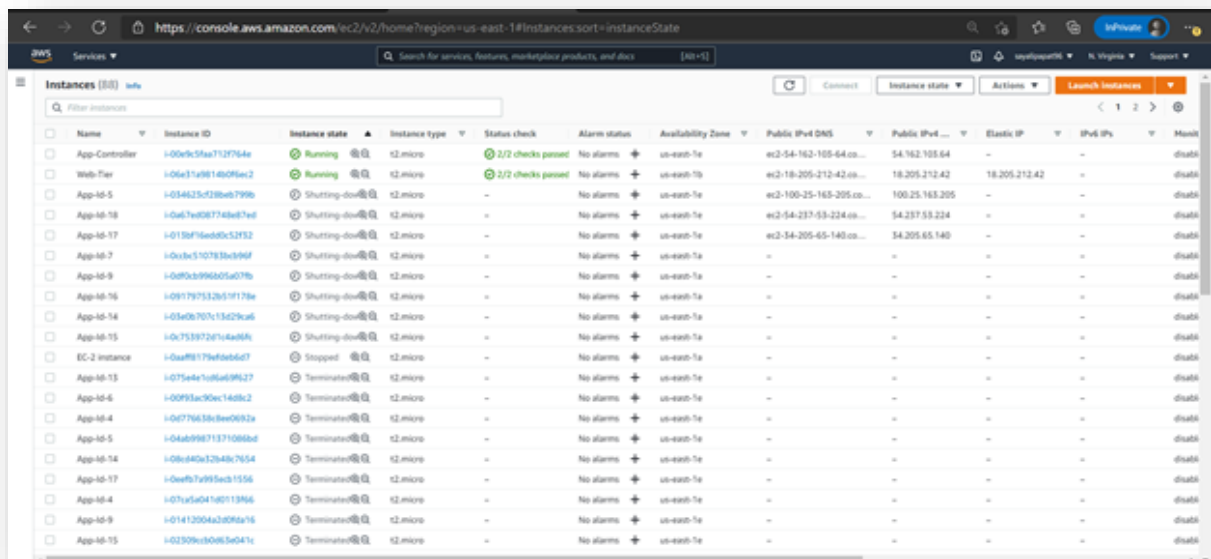
S3 input bucket:



Scaling up of instances for processing the images:

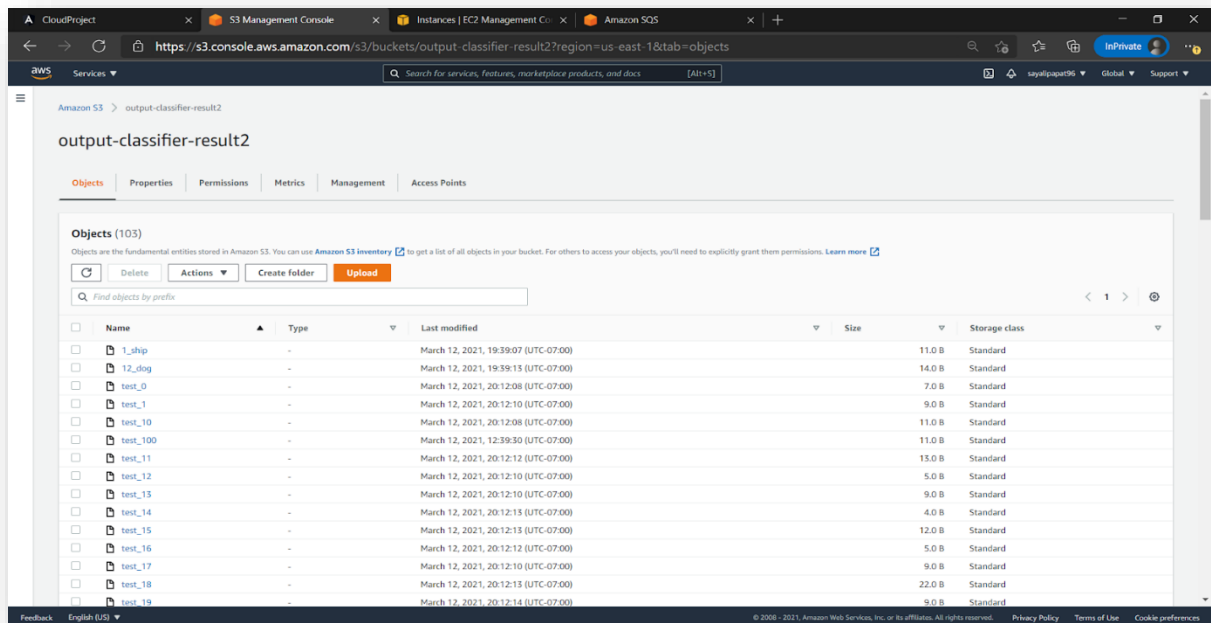


Scaling down of instances once the load is decreased:



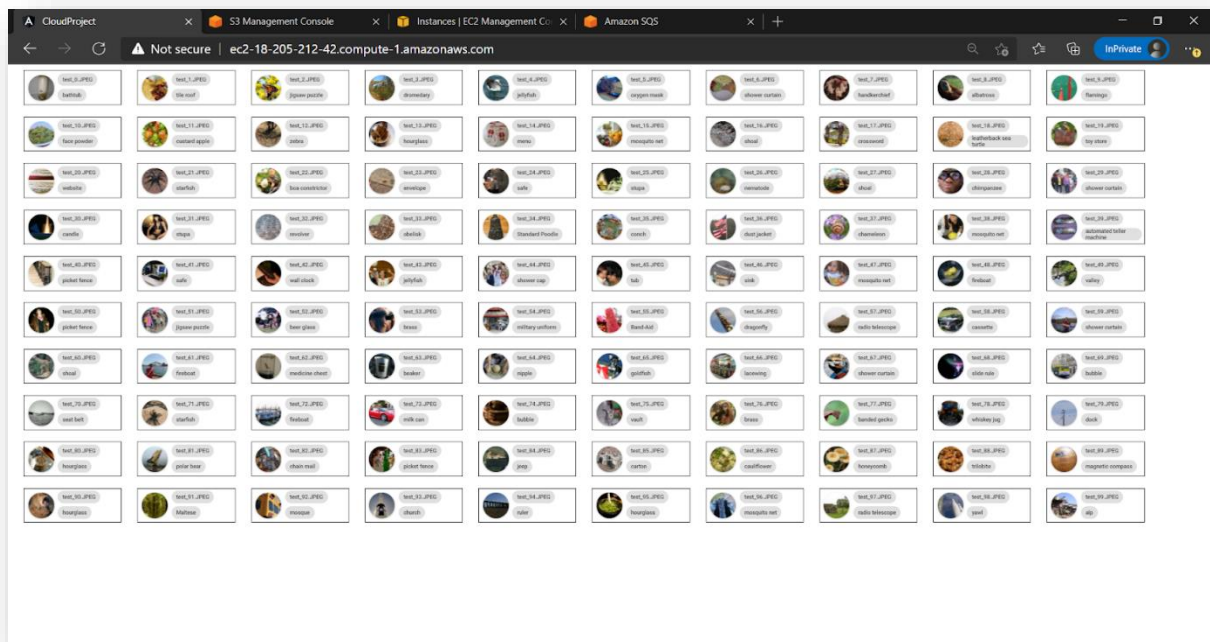
Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IP	Elastic IP	Monitoring
App-Controller	i-009c3f3a7127764e	Running	t2.micro	2/2 checks passed	No alarms	us-east-1e	ec2-54-162-105-64.co...	54.162.105.64	disabled
Web-Tier	i-06a37a9814b09f6e2	Running	t2.micro	2/2 checks passed	No alarms	us-east-1p	ec2-18-205-212-42.co...	18.205.212.42	disabled
App-10-5	i-034623c438b4b799b	Shutting-down	t2.micro	-	No alarms	us-east-1e	ec2-100-25-165-205.co...	100.25.165.205	disabled
App-10-18	i-0a67e9d8774b87e8d	Shutting-down	t2.micro	-	No alarms	us-east-1e	ec2-54-237-53-224.co...	54.237.53.224	disabled
App-10-17	i-0130f16e4d0c52f32	Shutting-down	t2.micro	-	No alarms	us-east-1e	ec2-54-205-45-140.co...	54.205.45.140	disabled
App-10-7	i-0a3e510783b3c96f	Shutting-down	t2.micro	-	No alarms	us-east-1a	-	-	disabled
App-10-9	i-0a90a899a050707b	Shutting-down	t2.micro	-	No alarms	us-east-1a	-	-	disabled
App-10-16	i-08179332b59f178e	Shutting-down	t2.micro	-	No alarms	us-east-1a	-	-	disabled
App-10-14	i-03a06707a13429ca6	Shutting-down	t2.micro	-	No alarms	us-east-1a	-	-	disabled
App-10-15	i-0c753972d714a68a	Shutting-down	t2.micro	-	No alarms	us-east-1a	-	-	disabled
EC-2 instance	i-0aa88179a6b6d607	Stopped	t2.micro	-	No alarms	us-east-1a	-	-	disabled
App-10-13	i-075e4e10b6a9627	Terminated	t2.micro	-	No alarms	us-east-1e	-	-	disabled
App-10-6	i-0093ac90ac14d8c2	Terminated	t2.micro	-	No alarms	us-east-1e	-	-	disabled
App-10-4	i-0a778638c8ee093a	Terminated	t2.micro	-	No alarms	us-east-1e	-	-	disabled
App-10-5	i-04ab99871371086ad	Terminated	t2.micro	-	No alarms	us-east-1e	-	-	disabled
App-10-14	i-08c40c32b43c7654	Terminated	t2.micro	-	No alarms	us-east-1e	-	-	disabled
App-10-17	i-0aefb7a995ac1556	Terminated	t2.micro	-	No alarms	us-east-1e	-	-	disabled
App-10-4	i-071a5a041a0713866	Terminated	t2.micro	-	No alarms	us-east-1e	-	-	disabled
App-10-9	i-01413004a3d06a76	Terminated	t2.micro	-	No alarms	us-east-1e	-	-	disabled
App-10-15	i-02309ec30463e041c	Terminated	t2.micro	-	No alarms	us-east-1e	-	-	disabled

Output S3 bucket:



Name	Type	Last modified	Size	Storage class
t_ship	-	March 12, 2021, 19:39:07 (UTC-07:00)	11.0 B	Standard
t2_dlog	-	March 12, 2021, 19:39:15 (UTC-07:00)	14.0 B	Standard
test_0	-	March 12, 2021, 20:12:08 (UTC-07:00)	7.0 B	Standard
test_1	-	March 12, 2021, 20:12:10 (UTC-07:00)	9.0 B	Standard
test_10	-	March 12, 2021, 20:12:08 (UTC-07:00)	11.0 B	Standard
test_100	-	March 12, 2021, 12:39:30 (UTC-07:00)	11.0 B	Standard
test_11	-	March 12, 2021, 20:12:12 (UTC-07:00)	13.0 B	Standard
test_12	-	March 12, 2021, 20:12:10 (UTC-07:00)	5.0 B	Standard
test_13	-	March 12, 2021, 20:12:10 (UTC-07:00)	9.0 B	Standard
test_14	-	March 12, 2021, 20:12:15 (UTC-07:00)	4.0 B	Standard
test_15	-	March 12, 2021, 20:12:13 (UTC-07:00)	12.0 B	Standard
test_16	-	March 12, 2021, 20:12:12 (UTC-07:00)	5.0 B	Standard
test_17	-	March 12, 2021, 20:12:10 (UTC-07:00)	9.0 B	Standard
test_18	-	March 12, 2021, 20:12:13 (UTC-07:00)	22.0 B	Standard
test_19	-	March 12, 2021, 20:12:14 (UTC-07:00)	9.0 B	Standard

Output displayed to the user:



2. Instance failure: We tested that if an EC2 instance fails to classify an image in the estimated time because of any reason, our monitor controller would terminate that instance and push the image URL in launch queue for re-processing
3. Faulty image: We tested with large size images for which the classifier will fail and will be stuck in a loop. We wrote a logic, which would handle and identify the failed image and update the user accordingly.

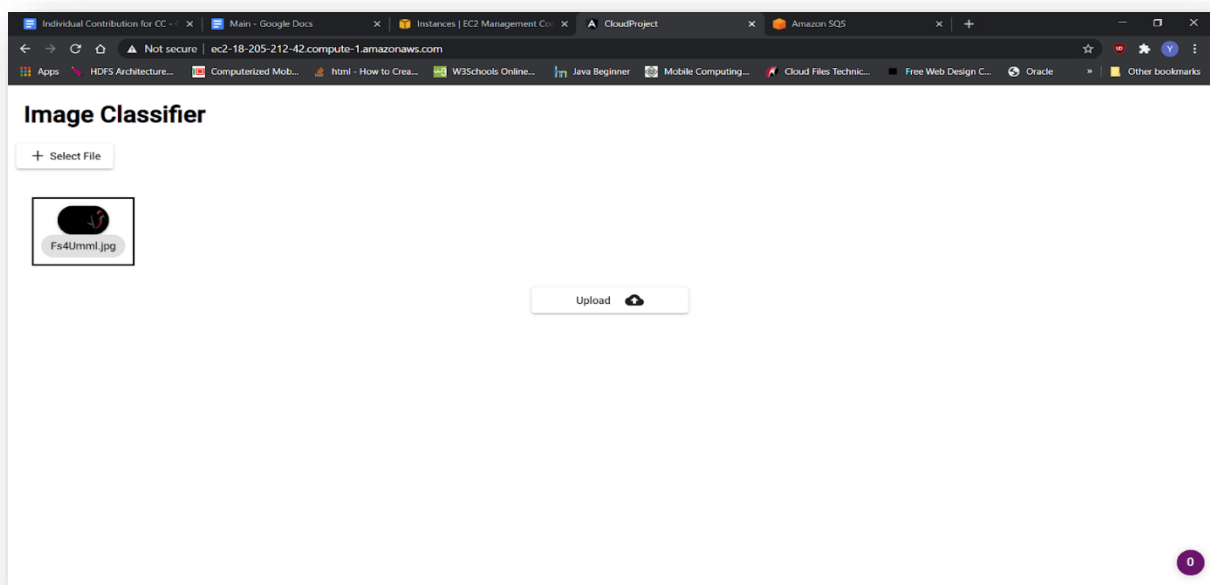
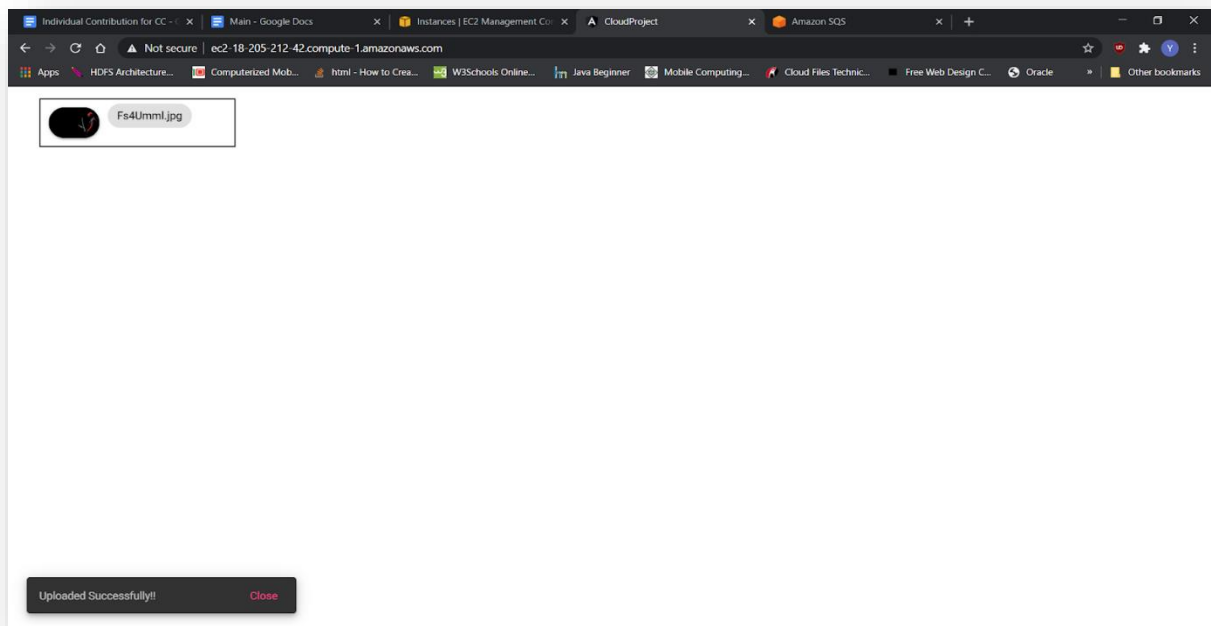
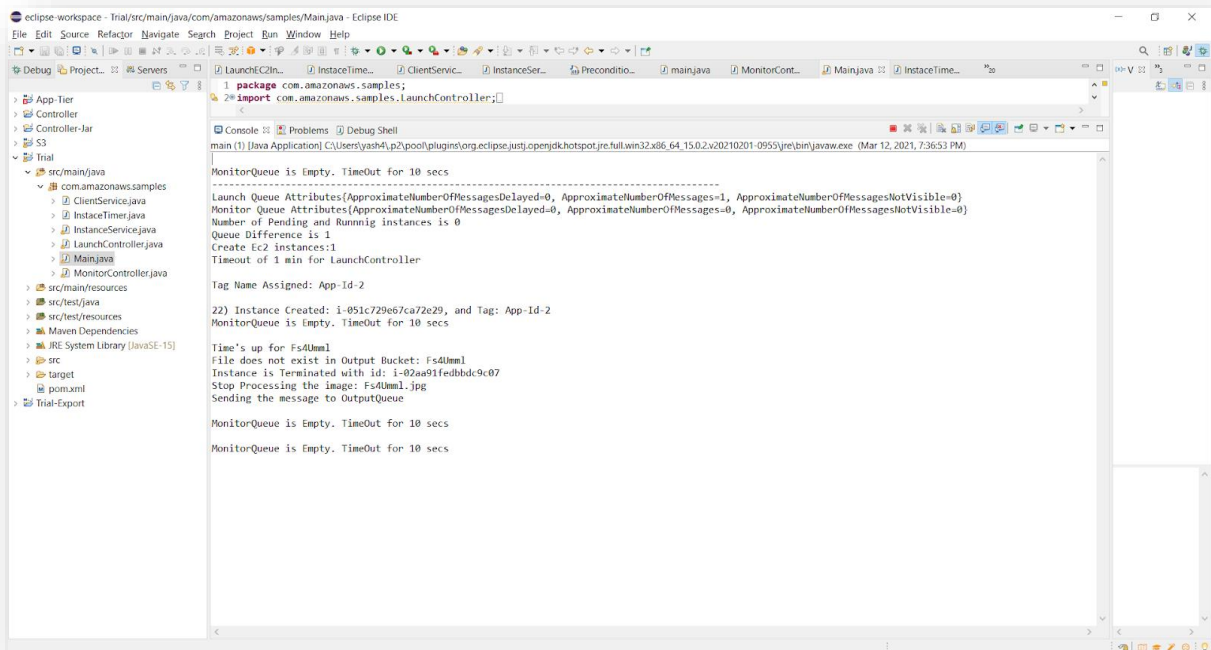


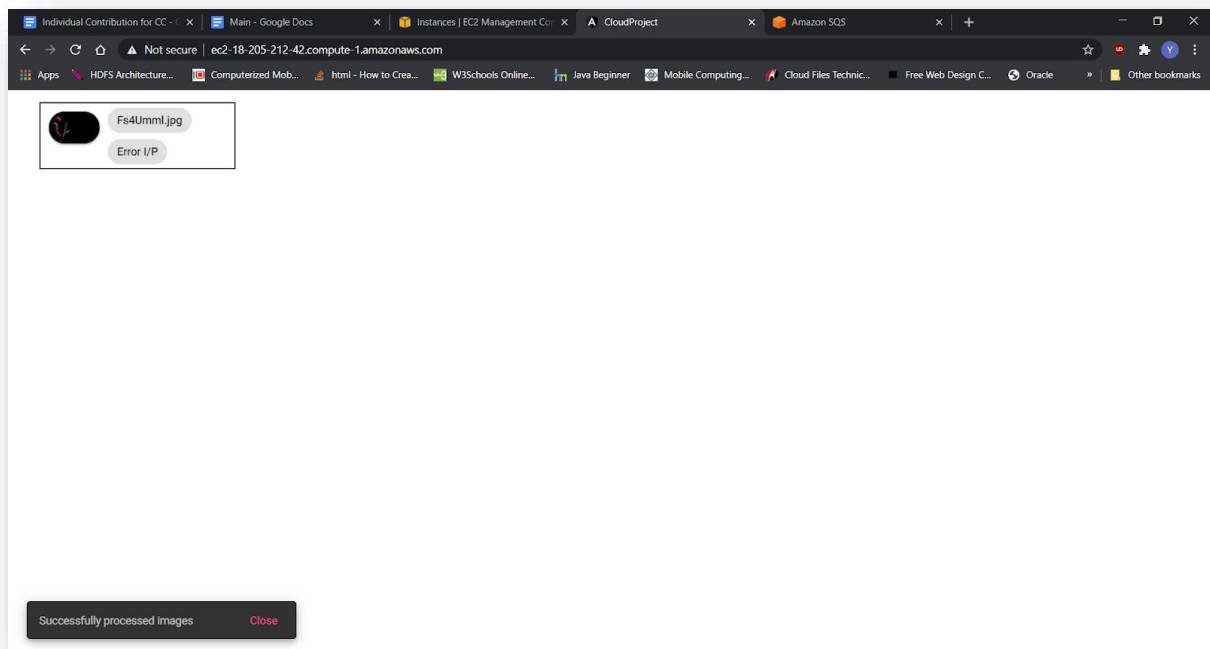
Image uploaded successfully



Logs from the controller



Output of Error Input Image displayed to the user



4. Multiple User uploading images: We tested our application with multiple users and verified that each user gets the results of the only those images which he had uploaded.
5. Tested for multiple images in different batches as follows:
 - (i) 1 image
 - (ii) 15 images
 - (iii) 50 images
 - (iv) 100 images
 - (v) 650 images
 - (vi) 15 images + delay of 1 min then 50 images

4. Code

App-Controller

1. Main.java:
 - a) The program starts its execution from the Main.java file.
 - b) In this file, we are executing the Launch Controller and Monitor Controller.
 - c) Both these controllers are processed in parallel with a thread for each controller.
2. ClientService.java:
 - a) Contains the details of all AWS clients used like S3, SQS, and EC2.
 - b) Authentication Credentials are declared in a private function and returned only with the class.
 - c) Creates S3 client, SQS Client, and S3 client using the Authentication credentials and returns required client by any other file.

3. InstanceService.java:
 - a) Helps in creating and terminating multiple EC2 clients.
 - b) fetchTagName()
 - (i) Fetches a unique tag name from a data set of App-Id-1, App-Id-2,, App-Id-18.
 - (ii) It keeps a track of Tag names that are in Pending and Running state for instances and helps in resolving duplication of any tag-name during the creation of a new one. Creates a new EC2 instance for the given AMI and adds a unique tag to it.
 - c) createInstance()
 - (i) Creates a new EC2 instance for the given AMI and adds a unique tag to it.
 - (ii) It also appends SecurityGroupId, UserData, TagSpecification, InstanceType, and ImageId while creating them.
 - d) createMultipleInstances()
 - (i) Creates multiple instances based on the number of requirements.
4. LaunchController.java:
 - a) Responsible for scaling up the application resources as per the demand of the requests.
 - b) countOfRunningAndPendingInstances()
 - (i) Returns the total number of Running and Pending instances
 - c) start()
 - (i) It contains a predefined count of maxNumberOfInstances allowed i.e. 18. We've kept it 18 as 2 instances are used for Web-Tier and Application-Controller.
 - (ii) At first, it fetches the total ApproximateNumberOfMessages of LaunchQueue and MonitorQueue
 - (iii) It then computes the difference of count of MonitorQueue and LaunchQueue
 - (iv) If the difference between them is greater than 0 and the count of running and pending instances is less than 18 then It can create more instances.
 - (v) It then computes the number of free instances
 - (vi) Based on the demand it Creates multiple instances and gives a wait of 1 min for long polling.
 - (vii) If there are no requests, it gives a wait of 10 secs.
 - (viii) The process would keep on repeating
5. MonitorController.java:
 - a) Responsible for monitoring the health of instances.
 - b) Keeps on checking the Monitor Queue after every 10 secs.
 - c) The Timeout of 10 secs is added for polling of messages after a specific time.
 - d) After a message is received it creates an InstanceTimer with all params for that object and deletes the message from the queue.
 - e) Instance Timer : Adds a new Timer for each object for 30 secs. After the Time is completed it checks for the Output in the S3 bucket and if it exists it doesn't do anything as the image has been classified perfectly. If the data doesn't exist in the Output S3 bucket it Terminates the instance and passes the data back to the Launch Queue to process it again. Now after each pass of the message to Launch Queue, a RetryCount value is checked which contains the number of times the image was processed. After it reaches a count of 3 i.e after 3 trials of an image on different instances we pass on the message to the Output Queue stating an Error in input the image and the result is displayed to the user

Web-Tier

It contains 2 components:

1. File-Upload:
 - a) Displays a button to Select Files. Once the files are selected they are stored inside onSelectFile().
 - b) Once the user clicks on the Upload button it calls the uploadFiles() function which passes the data to the uploadImageToS3(). Here the images are stored to the S3 bucket and on successful completion, it's passed to uploadToLaunchQueue() which passes the input image URLs to Launch Queue.
 - c) After successful completion of this the success the user is redirected to the Result component with a pop up stating images Uploaded Successfully.
2. Result:
 - a) Here the SQS Queue for Output is called and checked after every 10 secs limit if it's empty.
 - b) On the Html, there are images displayed waiting for the result. Once the Output queue gets a message it is passed on the output along with the animation class which triggers the flip of the image while displaying the output
 - c) The Output Queue is monitored until all messages are received and once they are done it stops monitoring the Output Queue.
 - d) After the process is completed it gives a pop-up stating "Successfully processed images"

EC2 Instance:

1. Image_classification.py
 - a) Responsible for classifying images and publishing the classified result to S3 output bucket.
 - b) Fetches the image URL from Launch Queue and sends it to Monitor Queue for monitoring with message status as running (i.e., the classifier is working on this image's classification). Once the image URL is fetched by an EC2 instance it deletes the message from the launch queue and sends it for monitoring, since the image is under process of classification.
 - c) Once the image is classified the result is sent to output S3 bucket and Output Queue.
 - d) EC2 instance polls the launch queue for the next image. If it is able to fetch the image it starts processing it and if it finds the launch queue empty, it considers all requested images are processed or are being processed by other instances and thus terminates itself to save the resources, since it has no work to do presently.

5. Individual contributions

Shivangi Singh (ASU ID: 1219123646)

I helped design the architecture of the system and contributed to the implementation of Fault Tolerance. I came up with the idea of a Monitor Controller. I, further, suggested implementation of an Output Controller in the architecture of a future model of the system. The Output Controller would have been responsible for receiving and deleting the output from the Output Queue, and sending the output further to the Web Tier. This would have offloaded the work of the Web Tier, which currently handles the output queue and fetches its information every 10 seconds.

Design:

Monitor Controller handles Monitor Queue, interacts with Launch Queue, Output Queue and S3 Output bucket and terminates EC2 instances if the specified condition is not met. Monitor controller receives messages from Monitor Queue and compares the input image with images stored in the S3 Output Bucket. If there is no match then the Controller terminates the Instance and sends the message back to Launch Queue.

Implementation:

I used Amazon Web Services modules in Java to create a Monitor Controller. Controller fetches data every 10 seconds from the Monitor Queue. Once it receives the message, it stores the attributes and the body of the message and then deletes it from the Monitor Queue. Now the timer is added to check the output bucket after the given time and if the output exists it means the images were classified successfully. If the image matched then no action was taken but if it did not match then a RetryCount value is checked 3 times. After that an error message is sent to the Output Queue and the result is displayed to the user.

Testing:

Unit Testing was performed on the Monitor Controller wherein I checked if the controller was able to receive messages from Monitor Queue and then delete it and then compare it further with S3 bucket. Further, I checked if for a faulty image, Monitor Controller was able to send back the message to Launch Queue and terminate the instance. Next, I monitored if the Monitor Controller is able to handle large amounts of messages and able to delete them and compare them within time. I also performed end to end testing wherein I checked if the system was able to handle large amounts of images, I monitored the queues and instances to see if the flow was working correctly. I further checked if more than one user was able to upload the images and the user gets the results of the only those images which he had uploaded.

Sayali Papat (ASU ID: 1219725975)

Design:

The design phase involved deciding the architecture and which service is to be used from AWS. What would be the role of each component and how they will communicate with each other. Me and my team discussed different models and finalized the architecture of the Project, with the goal of low latency for customers in mind. I worked on finalizing the design of Web Tier and dequeuing mechanism for the SQS launch queue which would send the images further for classification to the EC2 instances.

Implementation:

The implementation involved the creation of a web tier, app controller and the EC2 instances for managing and monitoring the processing as well as the result. I worked on the implementation of Web Tier using AWS-SDK for node. Operations like managing images uploaded to S3 input bucket, fetching the URL of S3 object, and sending it as a message to SQS launch queue, which is to be referred by the launch controller, fetching the result from the output queue once the image is processed were done by me. Created an AMI for EC2 instance which is being used for classifying the image. Worked on authorization with AWS S3 while accessing the input image from S3 bucket. Wrote a bash script which would change the necessary file permissions and carry the automatic execution of the image classification code. In the start we faced a lot of issues in implementing this, because of Linux user rights and file permissions, I debugged the access right issue and resolved it.

Setting up the AWS environment including the security groups, IAM user roles, managing the static IP for the Web Tier and the Application controller was done by me. I also wrote script which uses AWS CLI for checking the state of application controller and Web tier, which checks if the instances –are healthy or not and then either reboots the system which is having connectivity issues or create a new secondary instance, this can be considered as future scope of the application, supporting it with low downtime.

Testing:

Testing phase involved the testing of individual models and then integration testing. Me and my team performed end to end testing with different sizes of input image sets. I tested the Web tier functionality, which involved checking the correctness of the individual models and code correctness. While testing auto scaling logic, we encountered many corner case scenarios, to address that I optimized a few parameters and performed stress testing multiple times. In this testing and debugging phase, I helped in fixing the bug and improved performance. While testing the application we exhausted the AWS Free tier limit on one account, I migrated all the resources to a new account and integrated them for further testing.

Yash Ravindra Bhokare (ASU ID: 1219496304)

Design:

I was involved in understanding the project requirements and designing the initial architecture for scaling up and down the application resources based on the given load. I helped in considering the fault tolerance of the given EC2 instances. I considered different scenarios for multiple users and a need for adding a load balancer on the web-tier. Suggested a new UX design for the web application along with the run-time display of the output images.

Implementation:

Front-End - Implemented the design for the output page of the web-app, in which the images are displayed and flipped like a coin when output is received. Also implemented the dynamic collaboration of the output page and the typescript to display the image as soon as received. Connected the AWS-SDK with the Angular framework using node for the usage of SQS and S3. Resolved the cross-origin resource sharing (CORS) issue while trying to access the S3 bucket. Implemented the function which calls the Output Queue which keeps on checking messages after every 20 secs. Inserted a snack-bar that pops up when the output is received completely and stops the Output Queue requests. Dispatched the entire website on a Linux instance and hosted it using an apache server.

App-Controller - I've designed and implemented the Launch Controller. It handles the scaling up of the application resources based on the given demand. It never scales up more than the desired limit by keeping a track of running and pending instances. It's connected to the Launch and Monitor Queue to help detect the demand of the given request and the need for creation for more instances. I'm also creating a new instance or multiple instances based on the load. For each instance, I'm adding a unique tag-name that is specific to that instance and is not present in the currently running instances. I've made sure that no 2 instances of the same tag name will ever exist in the running state at one time. Once instances are created based on the demand from Launch Queue I'm giving a wait time of 1 minute to the Launch Queue; this will help in maintaining long polling. I've deployed the jar file on an EC2 instance of the Ubuntu System and created an AMI out of the given instance.

EC2 Classifier Instance - I've implemented the scale-down logic inside the python classifier. I've added connections to S3 and SQS using boto3 libraries. In this the classifier instance fetches messages from Launch Queue then deletes it and passes the message to Monitor Queue with params (status: running, instanceId, retry count). After each trial, the retryCount is increased by one. Then the classifier classifies the results and sends them to the Output bucket and queue to process further. It then checks for the load in Launch Queue and terminates itself if no message requests are remaining. So here the resources terminate themselves if the load is not much. Created a new AMI from this instance and used the AMI Id inside the Launch controller.

Testing:

I tested the Launch controller and its functionality for the creation of multiple instances which should never go beyond the given limit. I kept on uploading multiple images after a certain time to test the number of instances getting created after the poll time. I analysed the logs displayed during its processing, and at any point of failure, the logs helped to analyse the exact issue. I kept the controller on for about 5-6 hours for regression testing as it should never fail for any given scenario. Moreover, I checked the unique tag-names for each creation of instances. Furthermore, I tested the classifier to fetch data and terminate itself after an empty queue is given. I monitored the SQS queues for error states and checked the health of instances after an error has occurred. The stress test of the web application to check if correct results were returned and monitored the time for each image data set was conducted by me. At first, the Output page would run into an error stating insufficient resources as I was trying to access the Output queue continuously which ended up killing the web-page. This was resolved by adding a wait time of 10 secs and stopping the queue after all results were displayed. I've also tested the app in which input is given from multiple systems.