CSE 546 ---- Project Report

Group members:

S.No.	Name	Mail ID	Student Id
1	Venkata Divya Sai Gorijala	vgorijal@asu.edu	1225658473
2	Chandrika Kondapi	ckondapi@asu.edu	1223212406
3	Priyadarshini Ramakrishnan	pramak10@asu.edu	1225407339

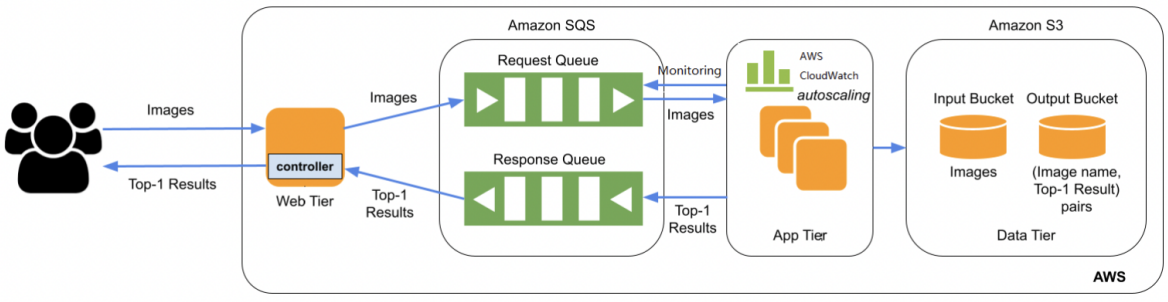
1. Problem statement

The main goal of the project is to build an elastic application that handles multiple requests concurrently. When the request demand increases the application should scale out and when demand drops it should scale in using cloud watch. The scaling should be done based on the depth request SQS queue using IaaS resources.

Developing an elastic application that can scale automatically in response to demand is critical for maintaining application resilience, reducing expenses, and improving user satisfaction. Automatic scaling enables the application to adapt to varying traffic levels and changing conditions in real-time. This approach can optimize resource usage, minimize expenses, and enhance application efficiency. Maintaining consistent response times, particularly during peak demand periods, is important to ensure users can access the application rapidly and efficiently. In summary, elastic scaling is an essential aspect of enhancing application performance and user experience.

2. Design and implementation

2.1 Architecture



Web Tier: Web Tier is the UI component for users to interact to upload images and return the results generated by App-tier. We are using python flask to upload images and to send and receive image-data and file-name to SQS. After all the uploaded images are sent to SQS then the response SQS queue is polled for the result and displaying the results to the user.The flask application is reverse proxied on an nginx server through gunicorn gateway interface in the AWS EC2 instance. In addition, for asynchronous processing of requests, an async worker called gevent was used with the flask application.

App Tier: After getting image-data from user sent through SQS, the app-tier decodes the byte data and converts it to image data and then uploads image-data and file name in the input S3 bucket , after labeling the image with the help of a given model it uploads the image_classification label and file_name to S3 result bucket .Simultaneously, the result is sent to response queue, for the results to be viewed by the users.

Autoscaling : Auto Scaling in AWS is a feature that allows you to automatically adjust the capacity of your EC2 instances based on the demand of your application. This helps ensure that your application can handle fluctuations in traffic and usage without the need for manual intervention.If the messages count is greater than 10 in sender SQS queue, cloud watch scale out alarm will be triggered and autoscaling group activity scale out will take place. If the messages count is less than 10 then cloud watch scale in alarm will be triggered and auto scaling activity scale in will take place.

Amazon SQS: Amazon Simple Queue Service (Amazon SQS) is a distributed message queue service provided by Amazon web services .It enables you to send, store and receive messages between different software components or microservices which can be running on different locations. Messages in SQS are stored in a queue until processed or deleted. Amazon SQS is used for decoupling the web tier and app tier. Amazon SQS offers two types of queues: standard and FIFO . We selected the standard queue due to its high throughput and best effort ordering.

S3 Buckets: Amazon S3 (Simple Storage Service) is a cloud-based object storage service provided by Amazon Web Services (AWS). S3 stores and retrieves any amount of data, at any time, from anywhere on the web. We created two S3 buckets for storing the images to store and retrieve them whenever needed.In first S3 input bucket file named 'cse546-project1-inputfiles' stores the image-data and file-name received from the SQS and other output bucket file named 'cse546-project1-results' stores the image classification labels and file name to send the data to SQS for sending it to user.

Cloud Watch: Cloud watch is one of the powerful tool for monitoring and observability in the AWS ecosystem, helping users ensure the performance, availability, and reliability of their applications and resources .We created two CloudWatch alarms which are used for auto scaling activity when the messages count increases the scale in and Scale out alarms will trigger and We are using the aws feature ApproximateNumberOfMessagesVisible for getting the messages count in SQS queue which will be assigned to autoscaling groups for scaling purpose.

2.2 Autoscaling

To ensure reliability and prevent issues with damaged EC2 instances from impacting scaling capabilities, the system uses event-based triggers for autoscaling. This is achieved by managing EC2 instances through Amazon Auto Scaling. The system always keeps the web tier and a minimum of one app tier instance of EC2 instances running. Additionally, two CloudWatch alarms, ScaleIn and ScaleOut, have been set up. An autoscaling group using a custom generated AMI for app-tier, and when demand increases, the CloudWatch alarms are triggered. Dynamic autoscaling policies have also been created within the autoscaling group, which activate after the CloudWatch alarms are triggered. If the ScaleIn alarm is triggered, the ScaleIn policy of the dynamic scaling policy in the autoscaling group is activated, and the desired capacity is reduced by three. On the other hand, if the ScaleOut alarm is triggered, the ScaleOut policy activity is initiated, and the desired capacity will be increased by three. The average of the SQS metric ApproximateNumberOfMessagesVisible is monitored over a period of one minute by cloudwatch to check if it's value is > or < 10. A value greater than 10 triggers the scale out alarm and a value less than 10 triggers the scale in alarm. Each alarm increases or reduces the capacity units by 3 units. This value of 3 was tuned to achieve the desired scaling behavior for the project. The cooldown period is set to 0 in our scenario to show auto scaling in action but ideally this cooldown period should be increased to reduce costs, as running an instance for several minutes is less expensive than terminating and creating a new instance every minute.

2.3 Member Tasks

Divya Sai Gorijala: My primary responsibility in the project is to set up the front-end application which is used to upload images and return the results that would be generated by the App-tier(Machine Learning model). There were many different design decisions to go with for the front end. My initial idea was to go with HTML, JavaScript as I have done front-end using them previously . But later through the thorough search found out that the Boto3 package in python is easy to use to send and receive data from Amazon Web Services components. As the setup of services was done, I deployed my code to EC2 instance. I have written a POST method which will take an image folder as an input and process all the images and send the image data to SQS. After the upload is complete it would send a message to created Simple Queue Service(SQS)with a body containing the file name image-data which would later be used by App-tier. Once all the images are sent to SQS then I am checking the response SQS for the result and printing the output. Once this was all set up I was able to run the flask application and test my code and be able to see the result.

Priyadarshini Ramakrishnan : My responsibility was to set-up the app-tier part , to integrate the app-tier in such a way that it can seamlessly retrieve images from the sqs queue sent by the web-tier, convert the byte data sent by the sqs queue into image data, store the images in s3 buckets, use the given image classification model to classify the images, upload the classification results in output S3 bucket and finally send the results to response SQS queue. I also created an AMI with the app-tier abilities which was further used to configure the launch template for the auto scaling group.I have also created a system service that enables the EC2 instances of the autoscaling group to invoke the python script (app-tier.py) during launching /

rebooting the system. I tested every functionality as mentioned above, using different number of inputs to the system, which was further extended with concurrent requests to the model.

Chandrika Kondapi: My main responsibility in this project is autoscaling . At first from AMI provided I created an autoscaling group for scaling the application. Before creating the auto scaling group i created a launch template ,where i have given the AMI .After creating launch template autoscaling group is created. Two SQS queues are created for sending and receiving messages between web tier and app tier ,two S3 buckets are created for storing the messages .A cloud metric is created using the Aws shell for getting the approximate number of messages in the SQS queue .I used cloud watch in order to generate alarms. I have created two alarms: scale in and scale out . Then inside the autoscaling group two dynamic scaling policies are created which are assigned to cloud watch alarms.

3. Testing and evaluation

Explain in detail how you tested and evaluated your application.

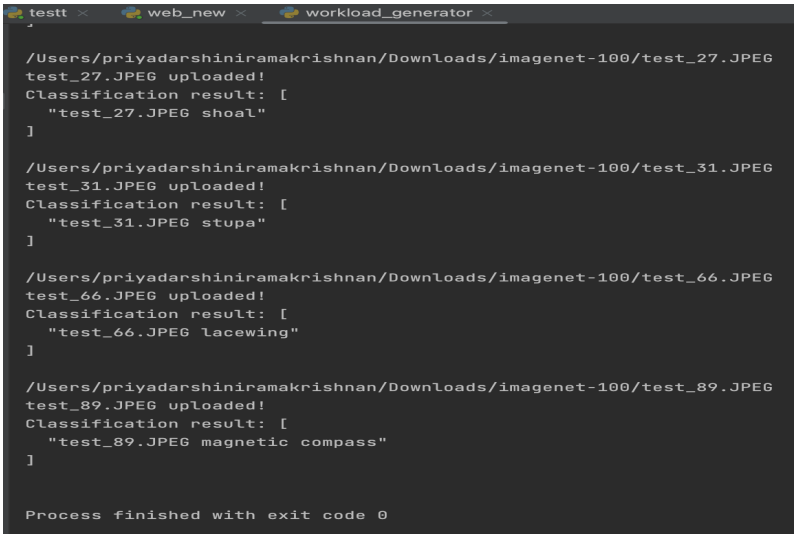
We tested our application using the UI we developed, TA's workload generator and finally multithread generator to test concurrent requests.

Workload generator: We first uploaded the images in small numbers, like 10 or 20 images at a time and tested the application. The POST requests were sent one by one by and the results were generated then and there using the workload generator. Hence the request queue had only one message at all times and there was no scope of autoscaling.

I/P to workload generator:

--num_request=10 --url="http://54.145.79.195/" --image_folder="/Users/priyadarshiniramakrishnan/Downloads/imagenet-100/"

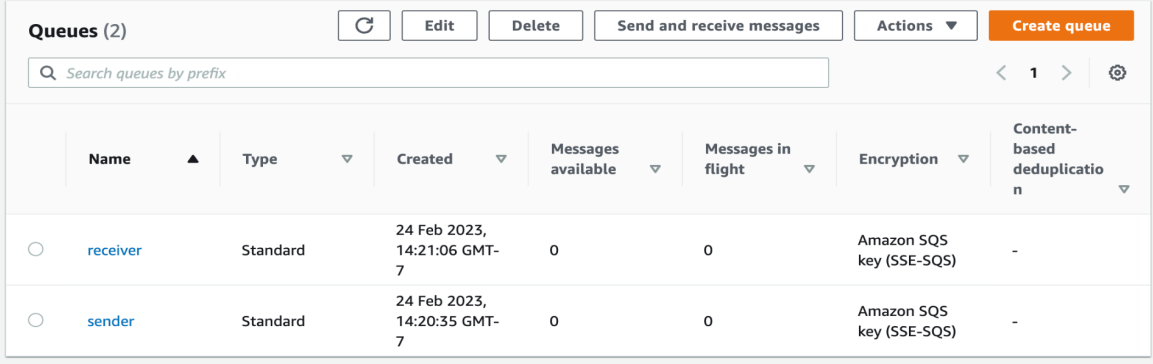
Image classification results:



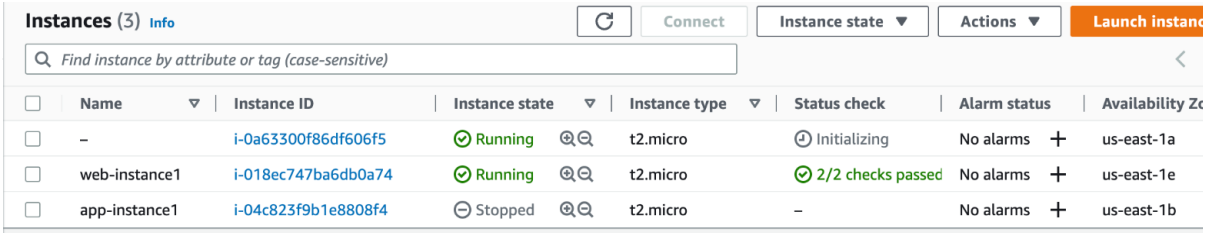
Processing Concurrent image requests:

Once the small scale testing was done, 100 images were concurrently uploaded using the multi thread executor provided by the TA's.

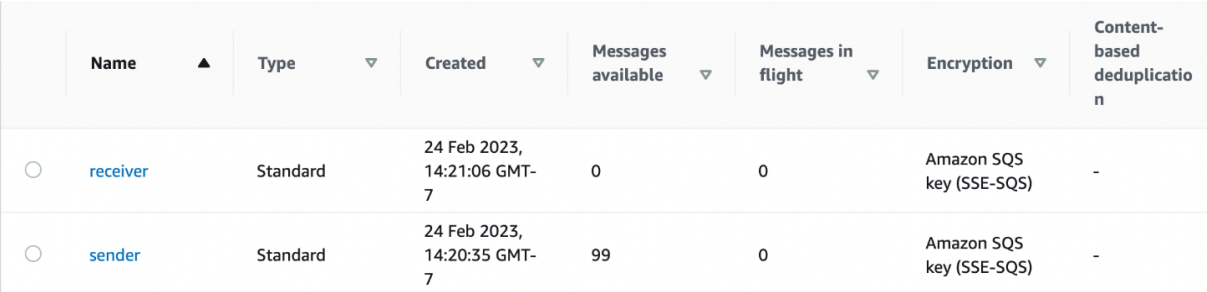
Queues having no messages initially:



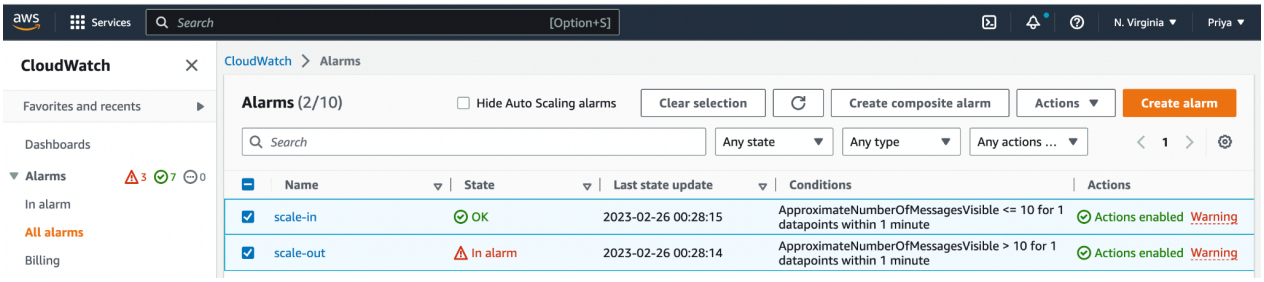
Running web-tier and minimum one app-tier instance from the ASG:



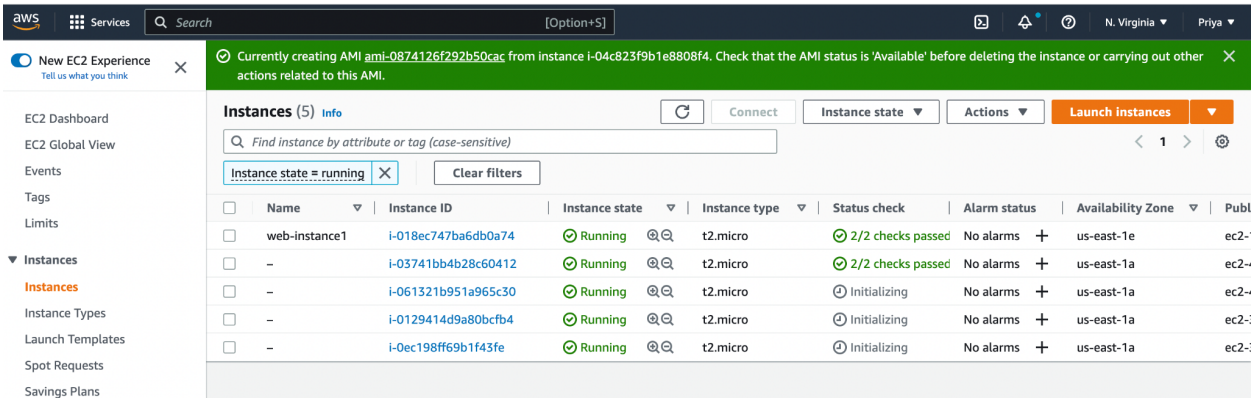
Request queue has 99 messages, uploaded concurrently through multi thread executor generator.



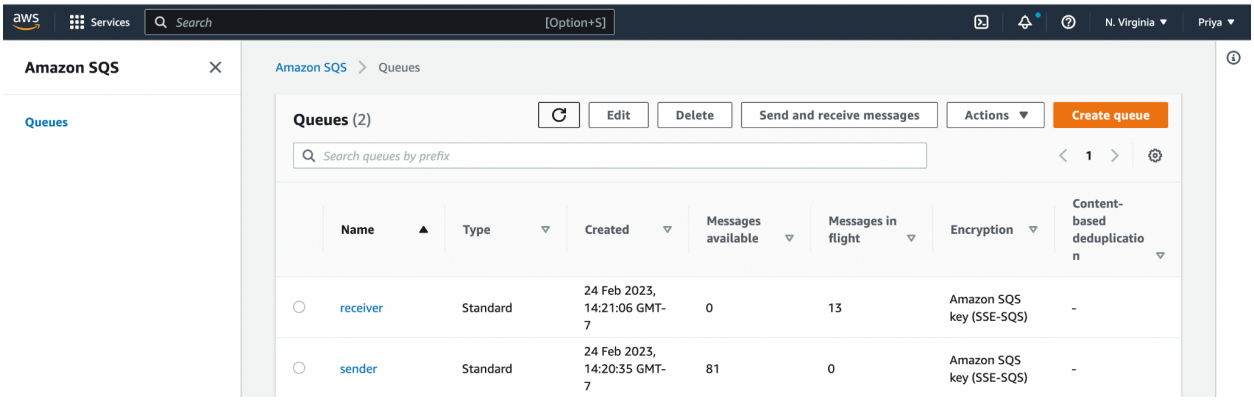
Cloud watch Scale-out alarm triggered.



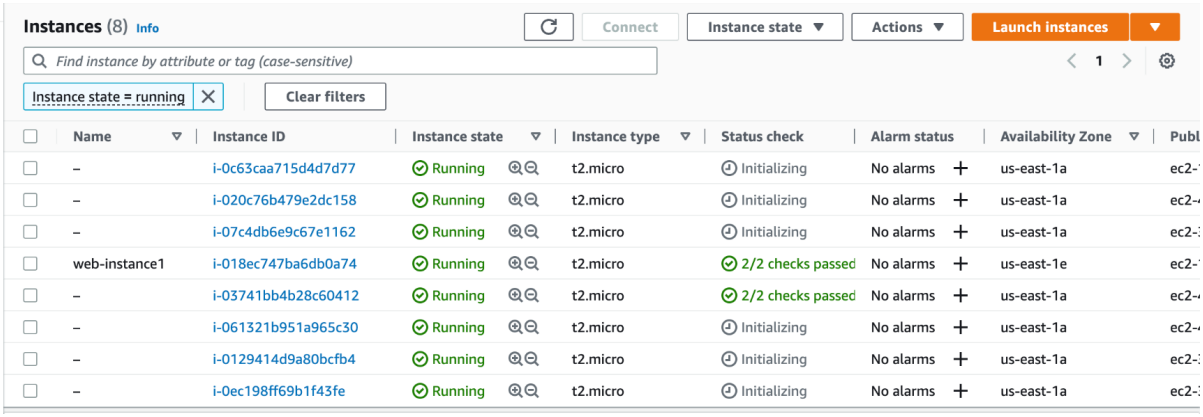
5 running instances after the first minute.

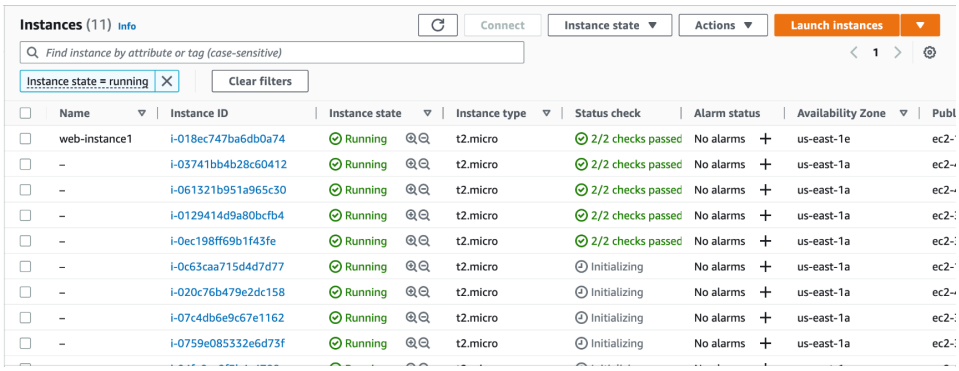


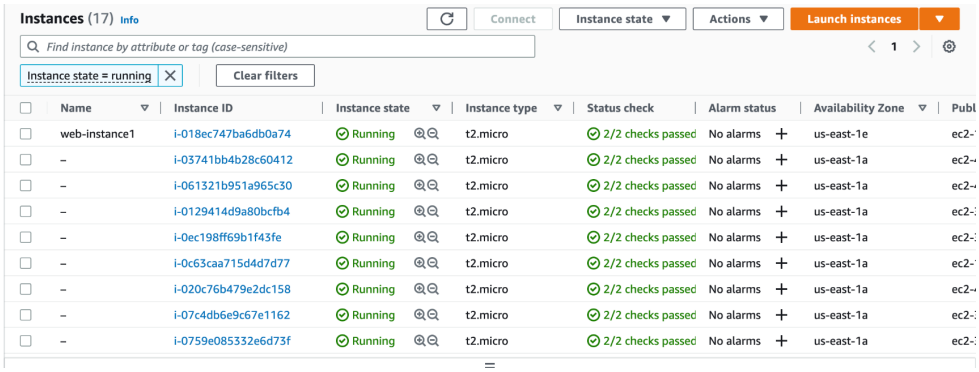
SQS messages reduced by 18 after first minute.



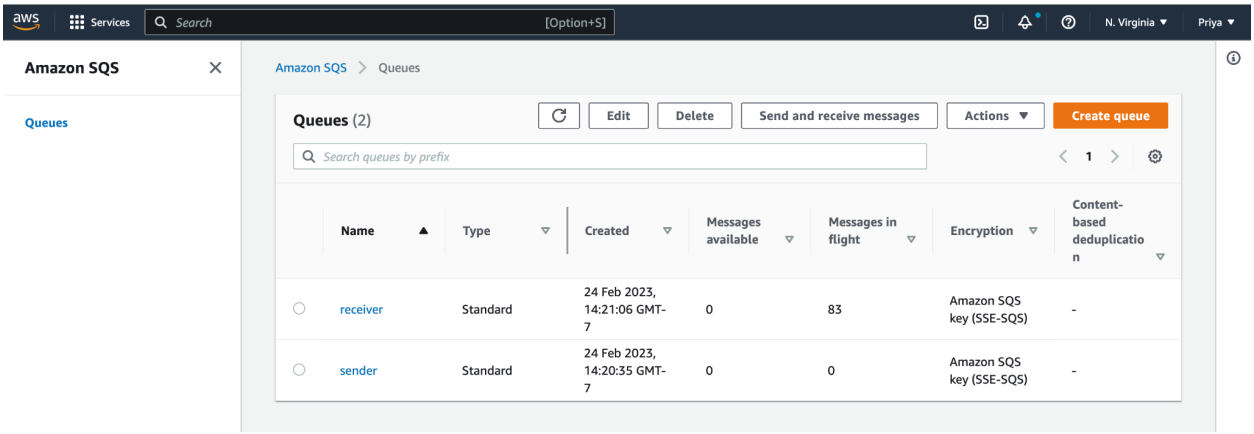
Eight running instances in the second minute, 11 in the 3rd minute…upto 17 instances.



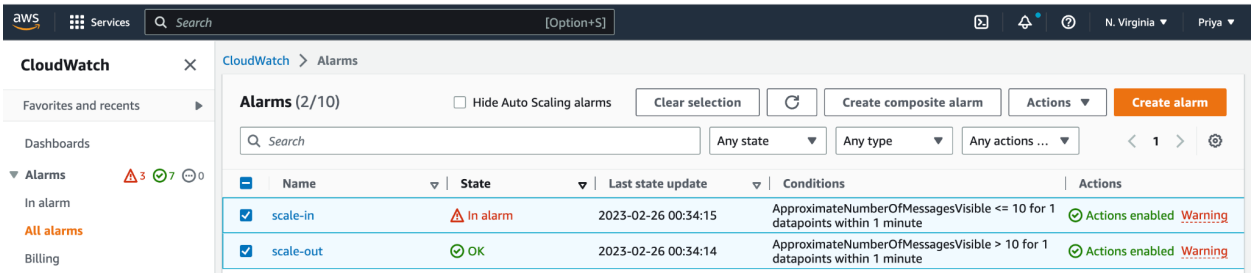




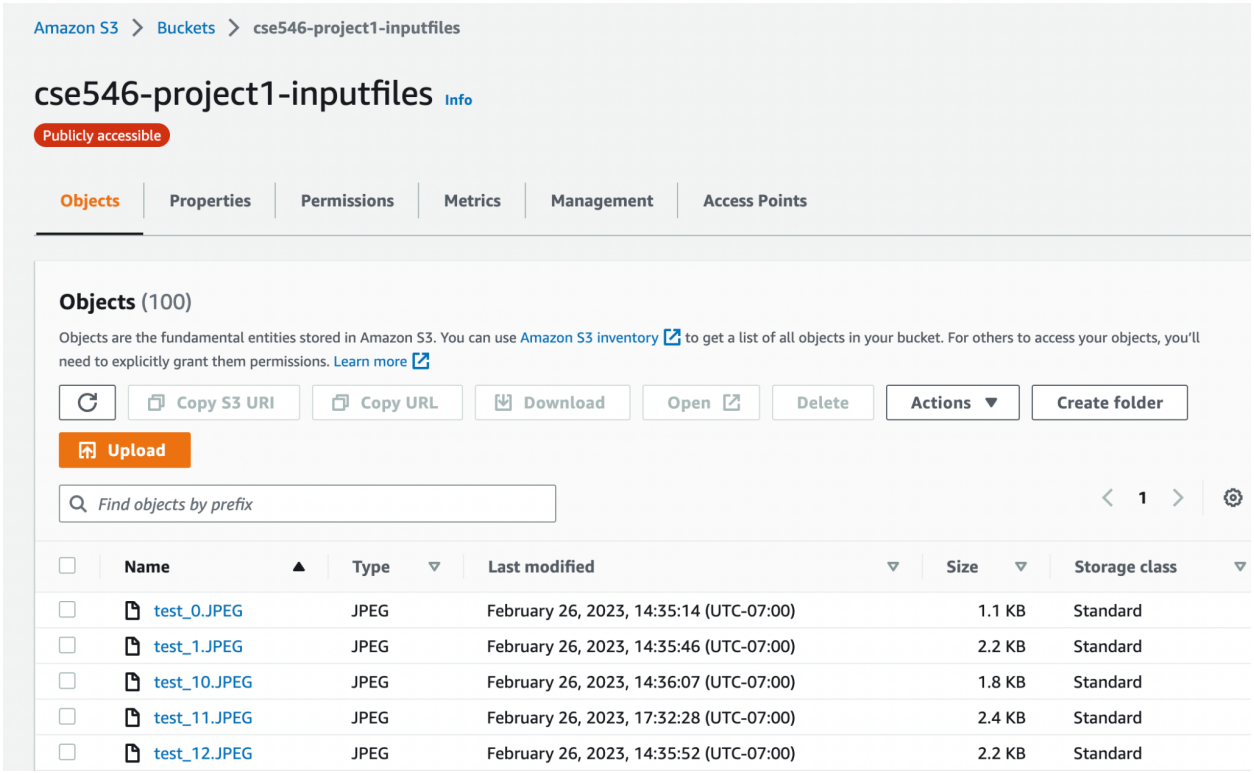
Request queue has zero messages to process:



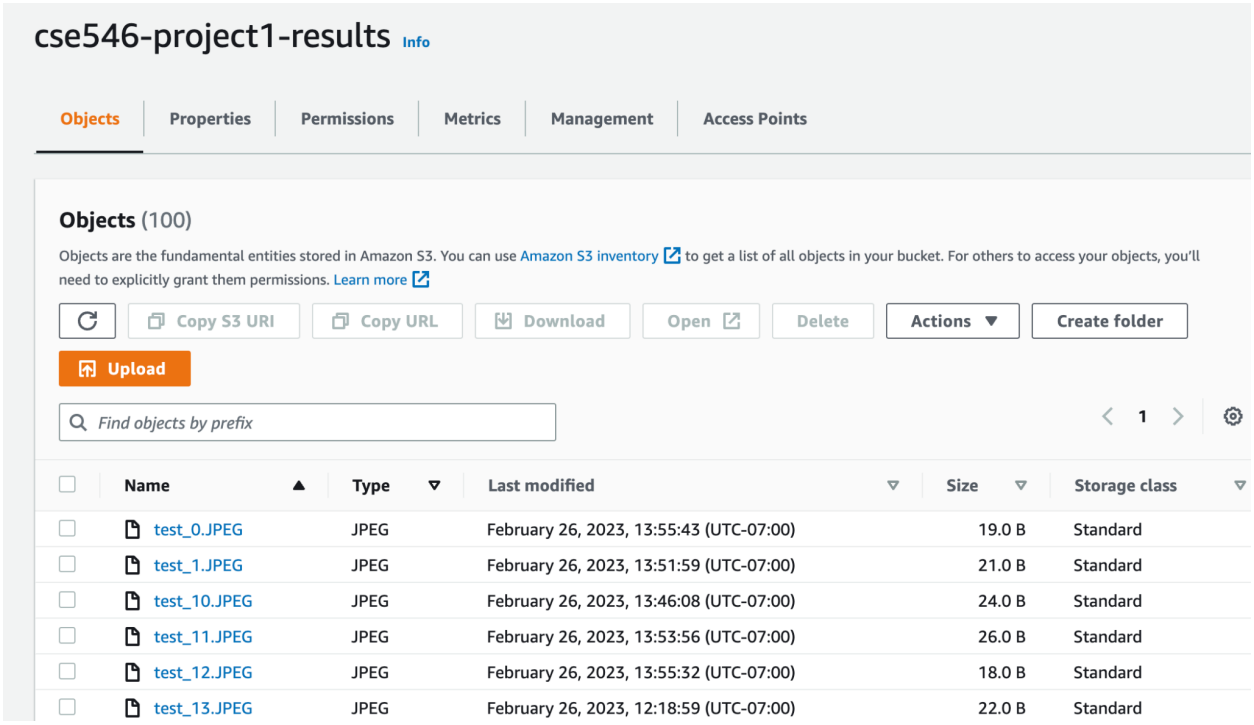
Scale-in alarm triggered.



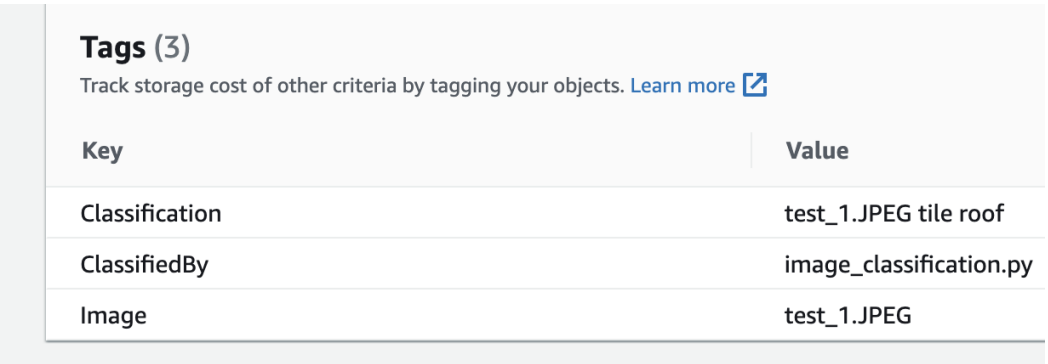
Input s3 bucket (100 objects):



Result S3 bucket(100 objects):



Tags enabled 👍

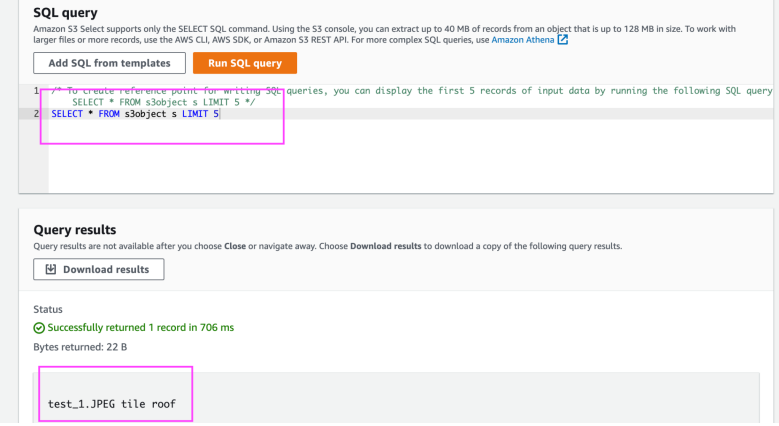


Tags (3)

Track storage cost of other criteria by tagging your objects. Learn more

Key	Value
Classification	test_1.JPEG tile roof
ClassifiedBy	image_classification.py
Image	test_1.JPEG

Query results:



SQL query

Amazon S3 Select supports only the SELECT SQL command. Using the S3 console, you can extract up to 40 MB of records from an object that is up to 128 MB in size. To work with larger files or more records, use the AWS CLI, AWS SDK, or Amazon S3 REST API. For more complex SQL queries, use Amazon Athena

Add SQL from templates | Run SQL query

```
1  /* To create reference point for writing SQL queries, you can display the first 5 records of input data by running the following SQL query
       SELECT * FROM s3object s LIMIT 5 */
2  SELECT * FROM s3object s LIMIT 5
```

Query results

Query results are not available after you choose **Close** or navigate away. Choose **Download results** to download a copy of the following query results.

Download results

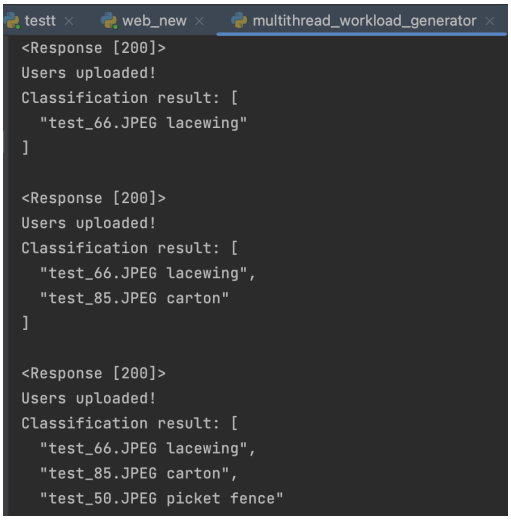
Status

Successfully returned 1 record in 706 ms

Bytes returned: 22 B

```
test_1.JPEG tile roof
```

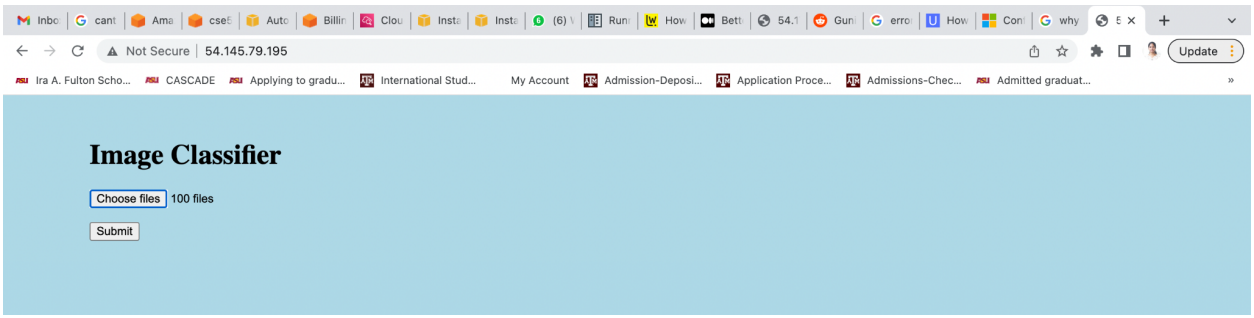
Multi- thread generator sample snapshot:



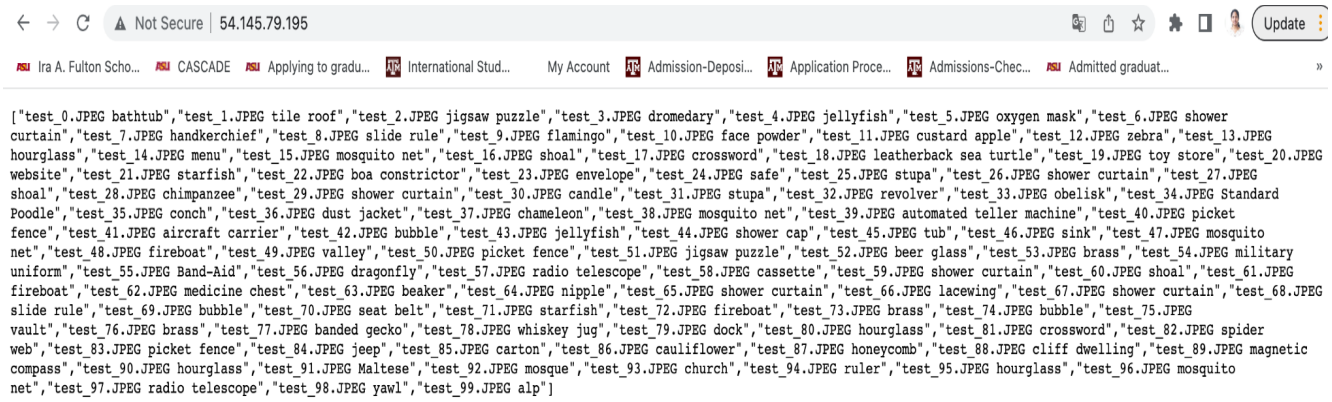
testt | web_new | multithread_workload_generator

```
<Response [200]>
Users uploaded!
Classification result: [
  "test_66.JPEG lacewing"
]

<Response [200]>
Users uploaded!
Classification result: [
  "test_66.JPEG lacewing",
  "test_85.JPEG carton"
]

<Response [200]>
Users uploaded!
Classification result: [
  "test_66.JPEG lacewing",
  "test_85.JPEG carton",
  "test_50.JPEG picket fence"
```

UI snapshot:



Classification results in browser:



["test_0.JPEG bathtub","test_1.JPEG tile roof","test_2.JPEG jigsaw puzzle","test_3.JPEG dromedary","test_4.JPEG jellyfish","test_5.JPEG oxygen mask","test_6.JPEG shower curtain","test_7.JPEG handkerchief","test_8.JPEG slide rule","test_9.JPEG flamingo","test_10.JPEG face powder","test_11.JPEG custard apple","test_12.JPEG zebra","test_13.JPEG hourglass","test_14.JPEG menu","test_15.JPEG mosquito net","test_16.JPEG shoal","test_17.JPEG crossword","test_18.JPEG leatherback sea turtle","test_19.JPEG toy store","test_20.JPEG website","test_21.JPEG starfish","test_22.JPEG boa constrictor","test_23.JPEG envelope","test_24.JPEG safe","test_25.JPEG stupa","test_26.JPEG shower curtain","test_27.JPEG shoal","test_28.JPEG chimpanzee","test_29.JPEG shower curtain","test_30.JPEG candle","test_31.JPEG stupa","test_32.JPEG revolver","test_33.JPEG obelisk","test_34.JPEG Standard Poodle","test_35.JPEG conch","test_36.JPEG dust jacket","test_37.JPEG chameleon","test_38.JPEG mosquito net","test_39.JPEG automated teller machine","test_40.JPEG picket fence","test_41.JPEG aircraft carrier","test_42.JPEG bubble","test_43.JPEG jellyfish","test_44.JPEG shower cap","test_45.JPEG tub","test_46.JPEG sink","test_47.JPEG mosquito net","test_48.JPEG fireboat","test_49.JPEG valley","test_50.JPEG picket fence","test_51.JPEG jigsaw puzzle","test_52.JPEG beer glass","test_53.JPEG brass","test_54.JPEG military uniform","test_55.JPEG Band-Aid","test_56.JPEG dragonfly","test_57.JPEG radio telescope","test_58.JPEG cassette","test_59.JPEG shower curtain","test_60.JPEG shoal","test_61.JPEG fireboat","test_62.JPEG medicine chest","test_63.JPEG beaker","test_64.JPEG nipple","test_65.JPEG shower curtain","test_66.JPEG lacewing","test_67.JPEG shower curtain","test_68.JPEG slide rule","test_69.JPEG bubble","test_70.JPEG seat belt","test_71.JPEG starfish","test_72.JPEG fireboat","test_73.JPEG brass","test_74.JPEG bubble","test_75.JPEG vault","test_76.JPEG brass","test_77.JPEG banded gecko","test_78.JPEG whiskey jug","test_79.JPEG dock","test_80.JPEG hourglass","test_81.JPEG crossword","test_82.JPEG spider web","test_83.JPEG picket fence","test_84.JPEG jeep","test_85.JPEG carton","test_86.JPEG cauliflower","test_87.JPEG honeycomb","test_88.JPEG cliff dwelling","test_89.JPEG magnetic compass","test_90.JPEG hourglass","test_91.JPEG Maltese","test_92.JPEG mosque","test_93.JPEG church","test_94.JPEG ruler","test_95.JPEG hourglass","test_96.JPEG mosquito net","test_97.JPEG radio telescope","test_98.JPEG yawl","test_99.JPEG alp"]

4. Code Explanation

All code with relevant READMEs can be found in the attached .zip file as part of the Project submission. There are two main python modules, app.py (web-tier) and app-tier.py (app-tier). Both web-tier and app-tier applications are written in Python Flask and Python, respectively.

- **Web-tier Installation:**
    - Move the app.py file to the project folder under the Ubuntu Linux server created as part of the EC2 web instance
    - Create a virtual Python environment using the below commands
        - *$ pip3 install virtualenv*
        - *$ virtualenv -p /usr/bin/python3 virtualenv_name*
        - *$ source virtualenv_name/bin/activate*
    - Install the necessary Python libraries in the active Python virtual environment using the below commands
        - pip3 install gunicorn
        - pip3 install gevent
        - pip3 install flask
        - pip3 install boto3

- Install Nginx in the EC2 web instance, an HTTP and reverse proxy server that routes the internet traffic to the Web gateway (Gunicorn) of our EC2 instance

    - *sudo apt-get install nginx*

- Create a systemd service (web-tier.service) for running the Gunicorn gateway during the boot of the EC2 instance with the below parameters

    [Unit]
    Description=Gunicorn instance for Iaas Group Project 1
    After=network.target network-online.target

    [Service]
    User=ubuntu
    Group=www-data
    WorkingDirectory=/home/ubuntu/project1
    ExecStart=/home/ubuntu/project1/venv/bin/gunicorn -b localhost:8000 --chdir /home/ubuntu/project1 app:app --timeout 30 -k gevent --worker-connections 1000 --workers 12
    Restart=always

    [Install]
    WantedBy=multi-user.target

- Modify the nginx default configuration with the below entries to enable port forwarding for HTTP (80), GET, and POST requests.

    # Default server configuration
    #

    upstream flaskapp {
    server 127.0.0.1:8000;
    }

    server {
    listen 80 default_server;
    listen [::]:80 default_server;

    root /var/www/html;

    server_name _;

    location / {
    proxy_pass http://flaskapp;
    proxy_read_timeout 900;
    proxy_connect_timeout 900;
    proxy_send_timeout 900;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }

- The Gunicorn and Nginx service is now set to start during the boot of the instance by executing the below commands

    - *sudo systemctl daemon-reload*
    - *sudo systemctl start web-tier.service*
    - *sudo systemctl enable web-tier.service*
    - *sudo systemctl start nginx*
    - *sudo systemctl enable nginx*

- The Web-tier instance is now ready to accept web requests from the internet

- **App-tier Installation:**

    - Move the app.py file to the project folder under the Ubuntu Linux server created as part of the EC2 App instance
    - Create a virtual Python environment using the below commands

        - *$ pip3 install virtualenv*
        - *$ virtualenv -p /usr/bin/python3 virtualenv_name*
        - *$ source virtualenv_name/bin/activate*

    - Install the necessary Python libraries in the active Python virtual environment using the below commands

        - *pip3 install boto3*
    - Create a systemd service (web-tier.service) for running the Python Application during the boot of the EC2 instance with the below parameters

        [Unit]
        Description=Python instance for the Image classification App
        After=network.target
        [Service]
        User=ubuntu
        Group=www-data
        WorkingDirectory=/home/ubuntu/project1
        ExecStart=python3 app_tier.py
        Restart=always
        [Install]
        WantedBy=multi-user.target

    - The Python Application is now set to start during the boot of the instance by executing the below commands

        - *sudo systemctl daemon-reload*
        - *sudo systemctl start app-tier.service*
        - *sudo systemctl enable app-tier.service*

    - The App-Tier instance is now ready and available to act as the backend for the requests triggered from the Web application.

Alternatively, the EC2 instances can be created from the AMIs provided as part of the .zip file submission.

app-tier.py:

Functionality: Receives raw Byte data of the image in the request SQS queue and convert it to image data and stores the image in the S3 bucket as objects for persistence.
I utilized the python library boto3 as the AWS SDK to interact with SQS queues and S3 buckets programmatically. The image is further processed using the image classification model provided to us, and the top-1 result of classification is then sent to the response SQS queue for consumption by the web tier. The classification results are also stored in output s3 buckets as objects for future reference, appropriate tags are also added to these objects.

Dependencies: Image_classification.py, which contains the Deep learning model, imagenet-labels.json . Copy the above files to your working directory and run app_tier.py. It keeps polling for messages in the request sqs queue and sends back the classified result to response sqs queue.

app.py

Functionality: upload images and file-name to request SQS queue. After all the uploaded images are sent to SQS then, the response SQS queue is polled for the result and displayed the results to the user.

Dependencies: Move WebMain.html to your working directory - which contains the template for UI. Run app.py, which starts a flask web server and renders our UI.

References

[1] Amazon Web Services. url: https://aws.amazon.com/.

[2] Boto3 documentation.
url:https://boto3.amazonaws.com/v1/documentation/api/latest/index.html

[3] https://medium.com/@jawad846/amazon-ec2-auto-scaling-884ea50d2d

[4] https://enlear.academy/sqs-with-auto-scaling-group-asg-e2cd30100cad

[5] https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-using-sqs-queue.html

[6] https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-available-cloudwatch-metrics.html