CSE 546 — Project1 Report

Group 29

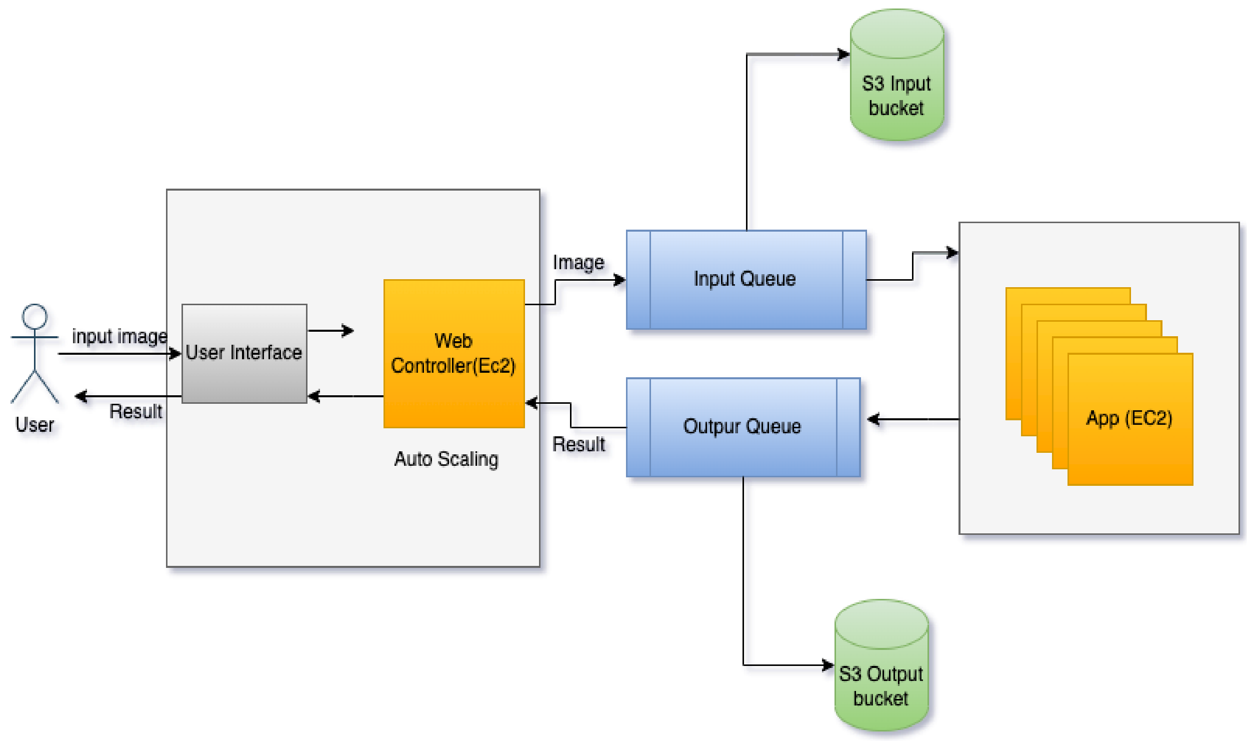
Amit Kumar Sinha
Deepakshi Mittal
Karan Ganeshdas Barhanpur

1. Problem statement

In this project, we try to build an elastic web application providing image recognition using Amazon Web Services IaaS resources. EC2 (AWS cloud resource for computing), S3 (AWS Object Storage Resources), and Simple Queue Service (AWS message queuing services) are used to build this application. Our application should provide correct image recognition results fast and cost effectively using the provided Deep Learning model. The application should also Scale In/Out depending upon the number of requests that need to be served.

2. Design and implementation

2.1 Architecture



2.2 Components Description

2.2.1 EC2(AWS Service)

AWS Elastic Computing Cloud is an On Demand , easily scalable Computing service in Cloud. In our project we use 1 EC2 instance to run the web- tier logic which is always running. 1-19 EC2 instances are used to classify input images based on the provided deep learning model. EC2 instances provide the scaling capacity to our application.

2.2.2 S3 (AWS Service)

Amazon Simple Storage Service S3 provides object level storage and persistence to our application. It is used by our application to store input image files and classification output in the form of key value pairs. It provides fault tolerance as images can be retrieved by a different EC2 instance in case one is not able to perform the task.

2.2.3 SQS(AWS Service)

Amazon Simple Queue Service is a scalable message queuing service. We use two queues namely InputQueue for queuing incoming requests and **OutputQueue** for queuing processed responses. It provides fault tolerance by decoupling the web and app tier.

2.2.4 Web Tier

Web Tier in our application consists of User Frontend to upload images, display results, an EC2 instance which listens for any new message in Input Queue. It also handles the crucial Scale-In, Scale Out logic based on the number of messages left to be served in Input Request Queue and accordingly Starts/Stop EC2 App Instances. It also stores input images files in S3 input buckets which are later accessed by the Face Recognition Model.

2.2.5 App Tier

App Tier in our application consists of EC2 instance(s) running the logic to classify the images using a model provided in face_recognition.py. New EC2 App instances are created from a pre-stored AMI which contains the executable jar file of AppInstance Java class, its dependencies and face_recognition model. It takes about 1-2 minutes for an instance to come to a running state. One EC2 instance serves one request at a time, stores the classification result in the S3 output bucket and output response Queue.

2.3 Autoscaling

Initially, we have a Web-Instance running on EC2 taking user request Scaling out of EC2 App-tier instances takes place in Web Tier. Total allowed EC2 instances according to project specification are 20. The maximum number of EC2 app instances that can be running at the same time is 19. So we used one for running a web tier which runs a thread to continuously check for new messages to be served in the InputQueue. We use the number of messages in the Input Queue and currently running instances as the decision parameter to spawn new instances.

App instances run in a loop to find some new request to be served in the Input SQS. If no requests are to be served are received in a predefined wait time, App instances will terminate i.e. Scaling In happens at Web Tier.

3. Testing and evaluation

All of the components were unit tested first to ensure that they are working as intended.

Frontend:

The frontend was tested locally before merging with the rest of the components,. The Request URL was tested to see if it was receiving images from the user, images are saved in the Input Queue and Input bucket.

Web Controller:

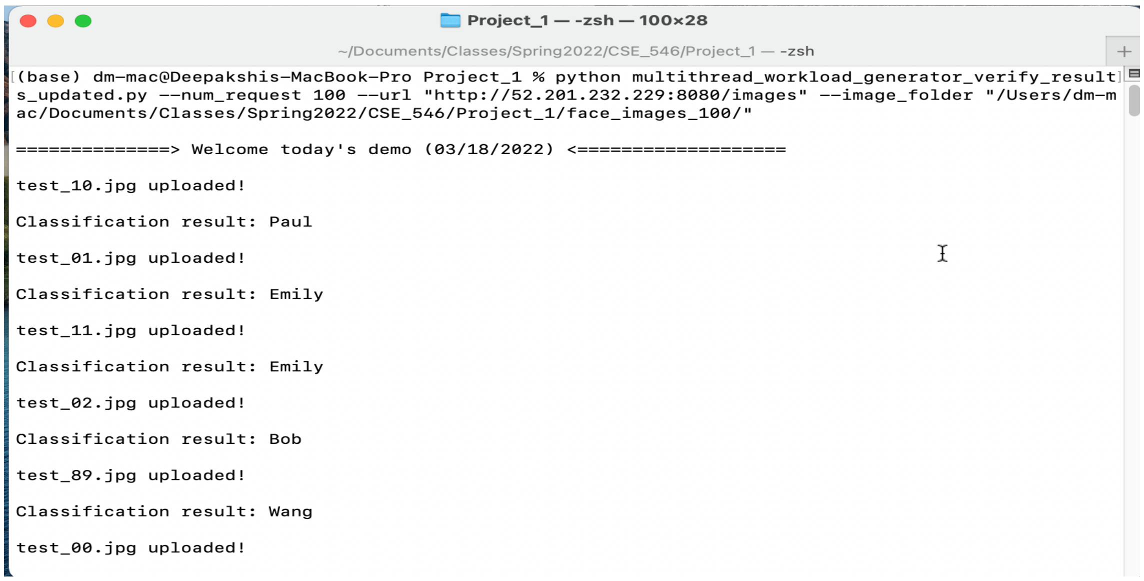
The web-controller is tested individually whether it is able to poll InputQueue for new messages. We created a dummy AWS SQS request queue that was constructed and populated with text messages. We checked if it was creating the correct number of EC2 instances based on the length of the request queue and the number of active instances. The new instances were checked to see if they were built using the given AMI. The maximum number of instances that could be produced (19) was also checked.

App Tier:

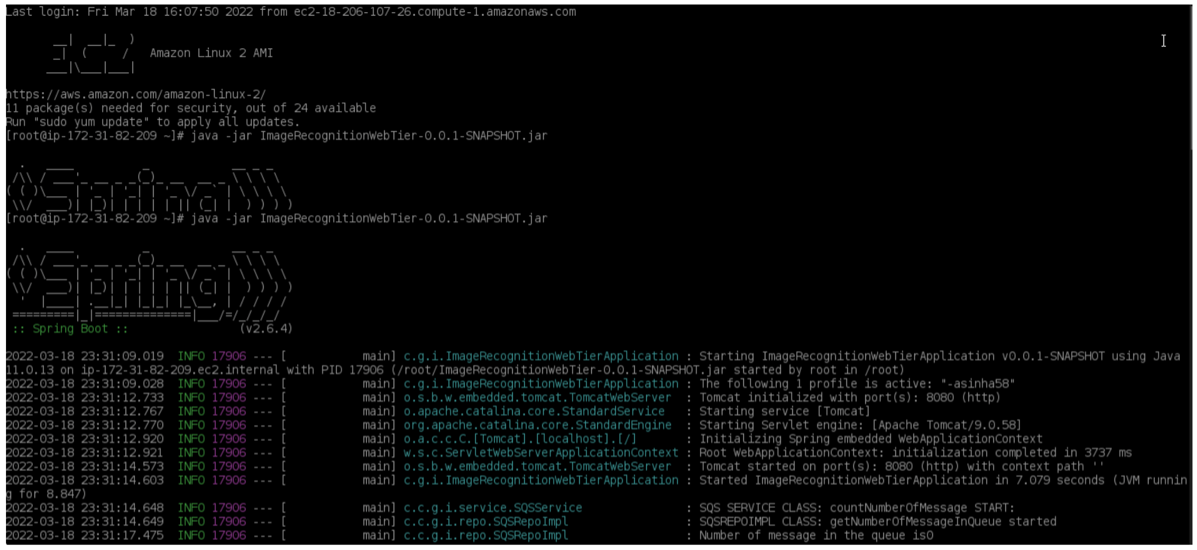
This component was checked to see if it correctly accommodated the requests from the request queue at the code level. The classification results from the deep learning model running in the app tier were checked against the expected output. Output Queue messages and Output Bucket Result were compared to the given expected output. Finally, the auto scale-in feature was verified by checking if the instances terminated when there were no more requests to serve.

After the integration, the entire project was evaluated using the given workload generator against 10 - 100 requests to ensure that the classification results matched the expected results. Finally, the AWS Console was used to evaluate EC2 instance scaling out and scaling in. SQS queues were monitored to ensure that messages were successfully saved, retrieved, and removed. S3 was checked to ensure that the input and classification results were saved correctly.

Screenshots:


Project_1 — -zsh — 100×28
~/Documents/Classes/Spring2022/CSE_546/Project_1 — -zsh

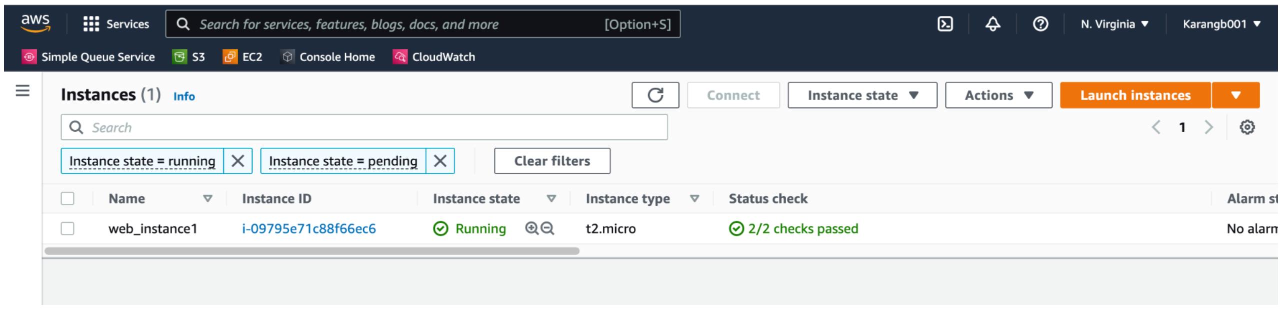
```
(base) dm-mac@Deepakshis-MacBook-Pro Project_1 % python multithread_workload_generator_verify_result
s_updated.py --num_request 100 --url "http://52.201.232.229:8080/images" --image_folder "/Users/dm-m
ac/Documents/Classes/Spring2022/CSE_546/Project_1/face_images_100/"

==============> Welcome today's demo (03/18/2022) <===================

test_10.jpg uploaded!

Classification result: Paul

test_01.jpg uploaded!

Classification result: Emily

test_11.jpg uploaded!

Classification result: Emily

test_02.jpg uploaded!

Classification result: Bob

test_89.jpg uploaded!

Classification result: Wang

test_00.jpg uploaded!
```



```
Last login: Fri Mar 18 16:07:50 2022 from ec2-18-206-107-26.compute-1.amazonaws.com

       __|  __|_  )
       _|  (     /   Amazon Linux 2 AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-2/
11 package(s) needed for security, out of 24 available
Run "sudo yum update" to apply all updates.
[root@ip-172-31-82-209 ~]# java -jar ImageRecognitionWebTier-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
[root@ip-172-31-82-209 ~]# java -jar ImageRecognitionWebTier-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.6.4)

2022-03-18 23:31:09.019  INFO 17906 --- [           main] c.g.i.ImageRecognitionWebTierApplication : Starting ImageRecognitionWebTierApplication v0.0.1-SNAPSHOT using Java
11.0.13 on ip-172-31-82-209.ec2.internal with PID 17906 (/root/ImageRecognitionWebTier-0.0.1-SNAPSHOT.jar started by root in /root)
2022-03-18 23:31:09.028  INFO 17906 --- [           main] c.g.i.ImageRecognitionWebTierApplication : The following 1 profile is active: "-asinha58"
2022-03-18 23:31:12.733  INFO 17906 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2022-03-18 23:31:12.767  INFO 17906 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2022-03-18 23:31:12.770  INFO 17906 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.58]
2022-03-18 23:31:12.920  INFO 17906 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2022-03-18 23:31:12.921  INFO 17906 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 3737 ms
2022-03-18 23:31:14.573  INFO 17906 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2022-03-18 23:31:14.603  INFO 17906 --- [           main] c.g.i.ImageRecognitionWebTierApplication : Started ImageRecognitionWebTierApplication in 7.079 seconds (JVM runnin
g for 8.847)
2022-03-18 23:31:14.648  INFO 17906 --- [           main] c.c.g.i.service.SQSService               : SQS SERVICE CLASS: countNumberOfMessage START:
2022-03-18 23:31:14.649  INFO 17906 --- [           main] c.c.g.i.repo.SQSRepoImpl                 : SQSREPOIMPL CLASS: getNumberOfMessageInQueue started
2022-03-18 23:31:17.475  INFO 17906 --- [           main] c.c.g.i.repo.SQSRepoImpl                 : Number of message in the queue is0
```

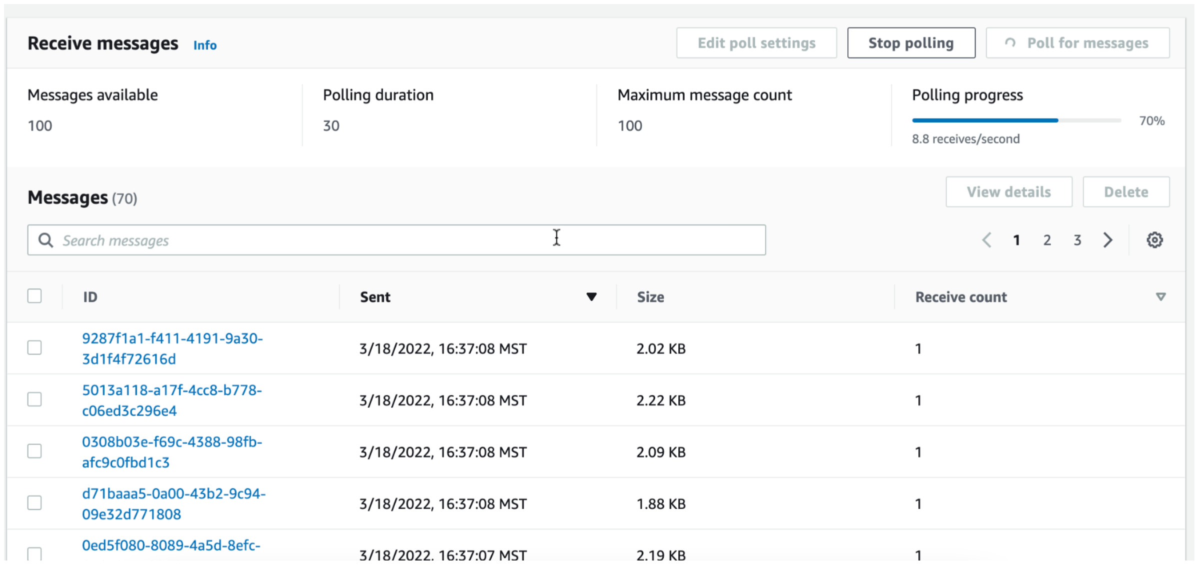
i-09795e71c88f66ec6 (web_instance1)

Public IPs: 52.201.232.229   Private IPs: 172.31.82.209



Instances (1) Info

	Name	Instance ID	Instance state	Instance type	Status check	Alarm st
	web_instance1	i-09795e71c88f66ec6	Running	t2.micro	2/2 checks passed	No alarm

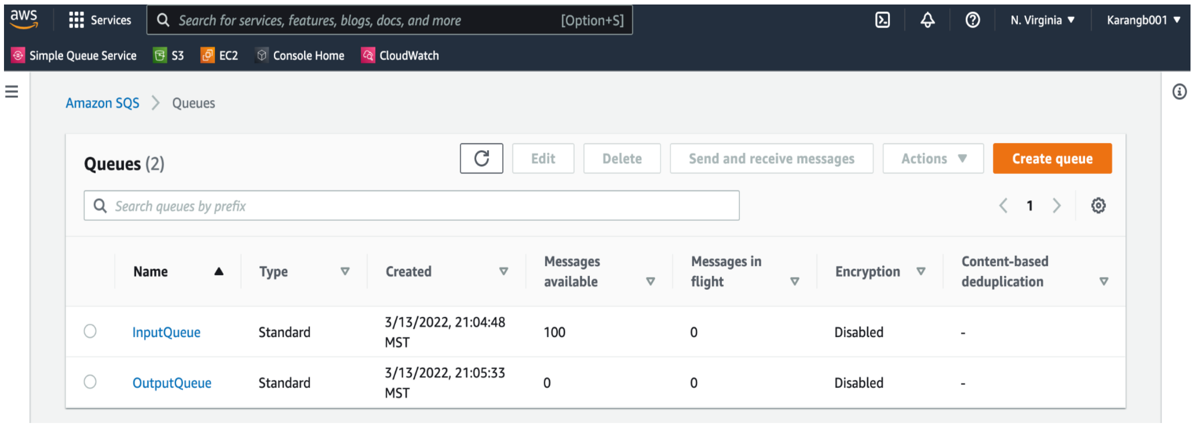


Receive messages Info

Messages available	Polling duration	Maximum message count	Polling progress
100	30	100	70% 8.8 receives/second

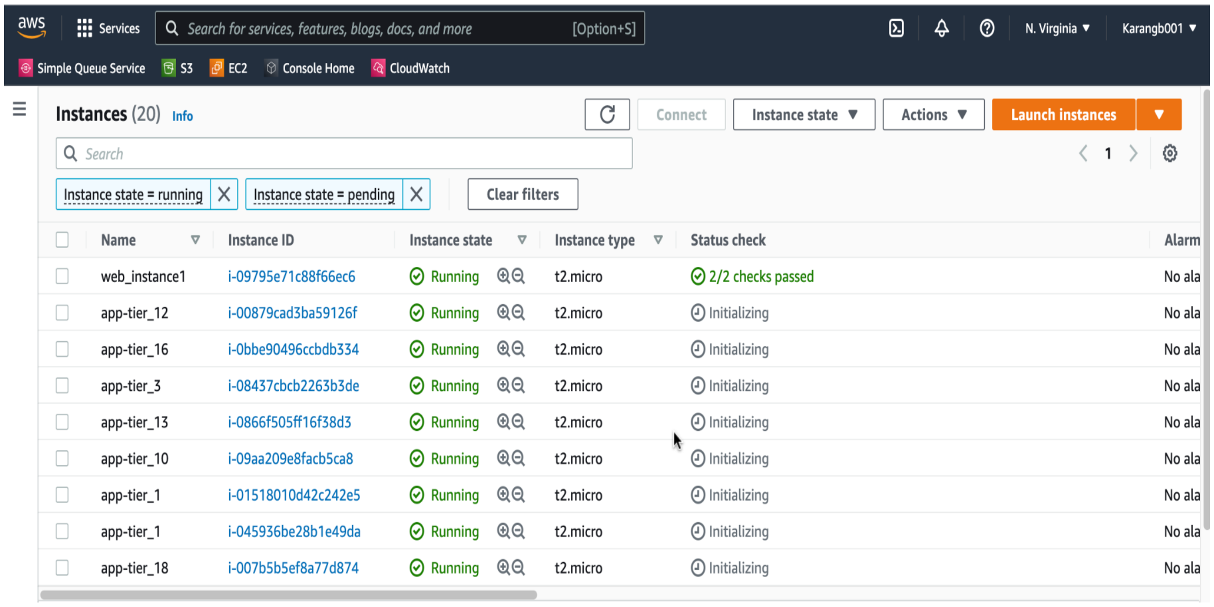
Messages (70)

	ID	Sent	Size	Receive count
	9287f1a1-f411-4191-9a30-3d1f4f72616d	3/18/2022, 16:37:08 MST	2.02 KB	1
	5013a118-a17f-4cc8-b778-c06ed3c296e4	3/18/2022, 16:37:08 MST	2.22 KB	1
	0308b03e-f69c-4388-98fb-afc9c0fbd1c3	3/18/2022, 16:37:08 MST	2.09 KB	1
	d71baaa5-0a00-43b2-9c94-09e32d771808	3/18/2022, 16:37:08 MST	1.88 KB	1
	0ed5f080-8089-4a5d-8efc-	3/18/2022, 16:37:07 MST	2.19 KB	1



Queues (2)

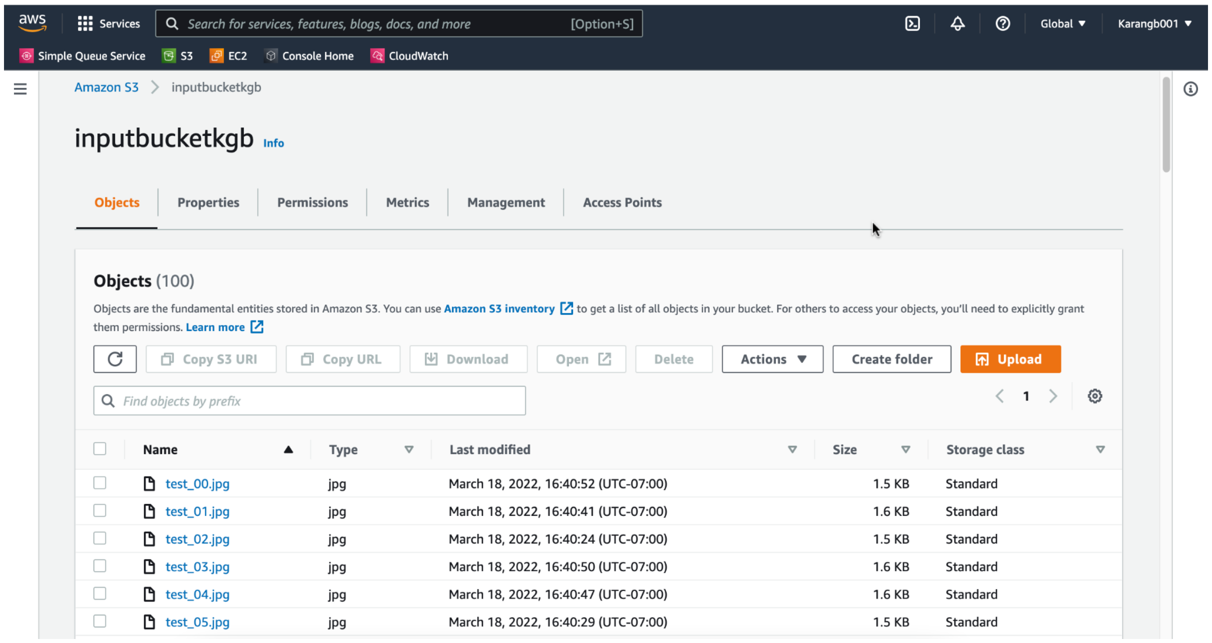
	Name	Type	Created	Messages available	Messages in flight	Encryption	Content-based deduplication
○	InputQueue	Standard	3/13/2022, 21:04:48 MST	100	0	Disabled	-
○	OutputQueue	Standard	3/13/2022, 21:05:33 MST	0	0	Disabled	-



Instances (20)

Instance state = running | Instance state = pending

	Name	Instance ID	Instance state	Instance type	Status check	Alarm
☐	web_instance1	i-09795e71c88f66ec6	Running	t2.micro	2/2 checks passed	No ala
☐	app-tier_12	i-00879cad3ba59126f	Running	t2.micro	Initializing	No ala
☐	app-tier_16	i-0bbe90496ccbdb334	Running	t2.micro	Initializing	No ala
☐	app-tier_3	i-08437cbcb2263b3de	Running	t2.micro	Initializing	No ala
☐	app-tier_13	i-0866f505ff16f38d3	Running	t2.micro	Initializing	No ala
☐	app-tier_10	i-09aa209e8facb5ca8	Running	t2.micro	Initializing	No ala
☐	app-tier_1	i-01518010d42c242e5	Running	t2.micro	Initializing	No ala
☐	app-tier_1	i-045936be28b1e49da	Running	t2.micro	Initializing	No ala
☐	app-tier_18	i-007b5b5ef8a77d874	Running	t2.micro	Initializing	No ala

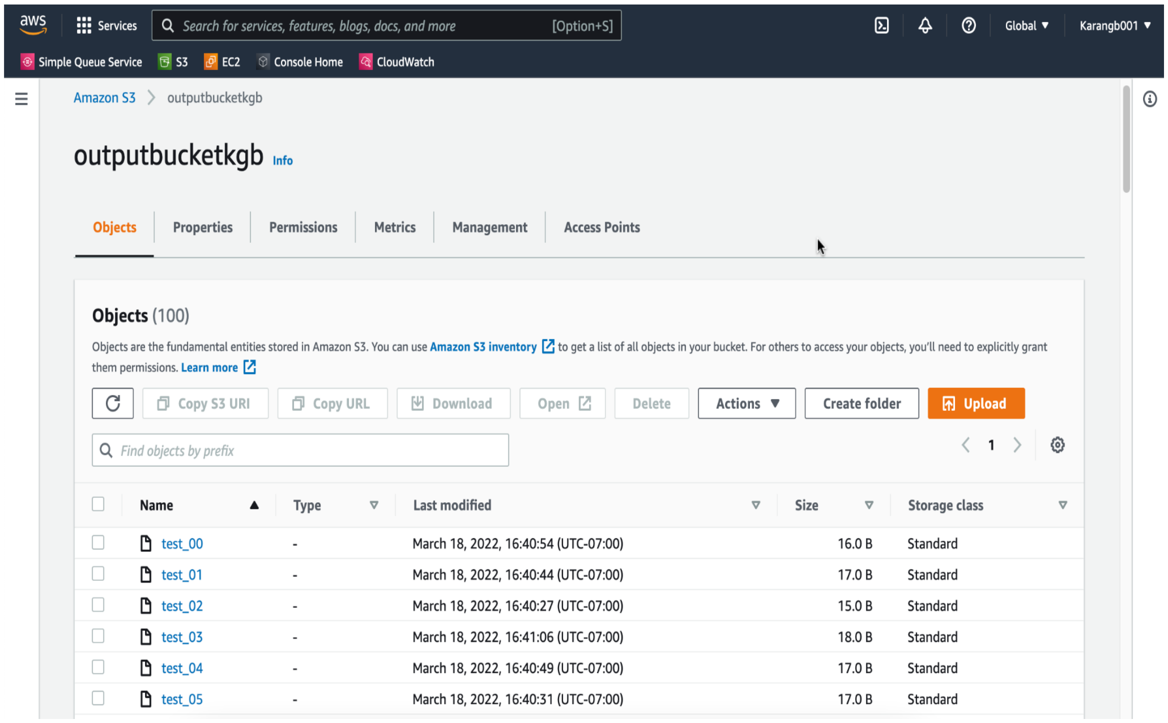


inputbucketkgb

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 18, 2022, 16:40:52 (UTC-07:00)	1.5 KB	Standard
☐	test_01.jpg	jpg	March 18, 2022, 16:40:41 (UTC-07:00)	1.6 KB	Standard
☐	test_02.jpg	jpg	March 18, 2022, 16:40:24 (UTC-07:00)	1.5 KB	Standard
☐	test_03.jpg	jpg	March 18, 2022, 16:40:50 (UTC-07:00)	1.6 KB	Standard
☐	test_04.jpg	jpg	March 18, 2022, 16:40:47 (UTC-07:00)	1.6 KB	Standard
☐	test_05.jpg	jpg	March 18, 2022, 16:40:29 (UTC-07:00)	1.5 KB	Standard

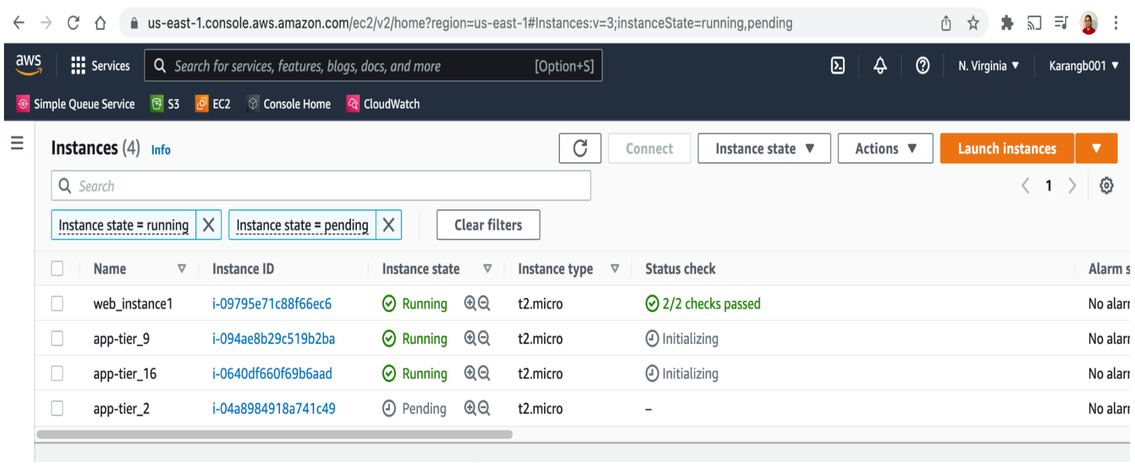

aws
Services
Search for services, features, blogs, docs, and more [Option+S]
Global
Karangb001
Simple Queue Service
S3
EC2
Console Home
CloudWatch
Amazon S3 > outputbucketkgb

outputbucketkgb Info

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	-	March 18, 2022, 16:40:54 (UTC-07:00)	16.0 B	Standard
test_01	-	March 18, 2022, 16:40:44 (UTC-07:00)	17.0 B	Standard
test_02	-	March 18, 2022, 16:40:27 (UTC-07:00)	15.0 B	Standard
test_03	-	March 18, 2022, 16:41:06 (UTC-07:00)	18.0 B	Standard
test_04	-	March 18, 2022, 16:40:49 (UTC-07:00)	17.0 B	Standard
test_05	-	March 18, 2022, 16:40:31 (UTC-07:00)	17.0 B	Standard


us-east-1.console.aws.amazon.com/ec2/v2/home?region=us-east-1#Instances:v=3;instanceState=running,pending
aws
Services
Search for services, features, blogs, docs, and more [Option+S]
N. Virginia
Karangb001
Simple Queue Service
S3
EC2
Console Home
CloudWatch

Instances (4) Info

Name	Instance ID	Instance state	Instance type	Status check	Alarm s
web_instance1	i-09795e71c88f66ec6	Running	t2.micro	2/2 checks passed	No alarn
app-tier_9	i-094ae8b29c519b2ba	Running	t2.micro	Initializing	No alarn
app-tier_16	i-0640df660f69b6aad	Running	t2.micro	Initializing	No alarn
app-tier_2	i-04a8984918a741c49	Pending	t2.micro	-	No alarn

4. Code

Web Tier:

1. Rest controller(ImageRecognitionController.java) : This Java class has the POST request for accepting the images from the user.

2. Send message to the input queue and receive output from the output queue: The SQSRepo and SQSService class provides this functionality. The application continuously listens(ScaleService) to the input and output queue to provide quick response to the user.

3. A hash table is implemented to provide a caching mechanism which is done to reduce the time for the user and provide a pleasant experience. The hash table is implemented in the ImageService class.

4. Finally, the output image name is returned from the controller class itself as a HTTP response along with the status code.

App Tier:

It consists of an AppInstance Java class that handles the logic. An image name is sent with each image request. This class gets the image object from the S3 Bucket and passes it to the deep learning model.. This model assigns a categorization to the image and returns it. This output is saved in S3 output bucket for persistence and is also sent as a message in the output in the SQS response queue. In the Response queue, it is sent as (image_name, classification_result). In S3 output bucket object name is image_name and content of file is the classification_result.

1. main(): The main function parses the command-line inputs. It also calls a polling mechanism on the input queue.

2. checkNewRequest(): It monitors the InputQueue on a regular basis and implements logic to terminate instances if there are no more requests to be served.(Scale In)

3. runDeepLearningModel(): This function accepts an image name as an argument. It retrieves the appropriate picture from the S3 input bucket and applies the deep learning model to it. Finally, the categorization output is returned.

4. enqueueResultInSQS(): Saves the categorization result in the Output Queue.

5. storeResultInS3(): Stores the classified images's results in the S3 output bucket for persistence

6. deleteRequestFromSQS(): deletes the processed message from Input Queue.

7. terminateInstance(): terminates the instance with a given InstanceId.

Installation Instructions:

Explain in detail how to install your programs and how to run them.

The app-frontend and app-controller are housed in the web-tier. ImageRecognitionWebTier is a jar file that must be opened and executed.

Steps:

Step1: Navigate to the directory where we have the ImageRecognitionWebTier jar file.

- cd /home/ec2-user/

Step2: Run the jar file with the following command.

- java -jar ImageRecognitionWebTier-0.0.1-SNAPSHOT.jar

This jar is where all of the dependencies are stored. As a result, we don't need to download the dependencies individually.

5. Individual contributions

Amit Kumar Sinha

This project involving the design of an Image Recognition Service, provided me deeper insights into Cloud Computing, and I gained hands-on experience with using AWS and its services like SQS, S3 and EC2. This project was a great learning experience and I would like to thank the professor, Ming Zhao, for giving me an opportunity to work on this project.

Design

Once I got the problem statement for the project and the architecture of the application, I decided to work on Web Tier and app scaling logic. Going with the architecture, I decided about the APIs needed to interact with all the AWS services. For the web tier, I used the AWS services like EC2 and SQS. I also came up with the design of the scale out logic. I designed an infinite loop for continuously listening to the number of messages in the input queue. This logic for the input queue was used in spawning the correct number of instances based on the traffic(number of messages in the input queue). I came up with the design of counting the number of EC2 instances in pending/running state, and consequently count the number of messages in the input queue and scaled the application based on these two values.

Implementation

After having a clear picture of the design of the application, I went ahead and implemented the design. I decided to work on Spring Boot and Java. I went through the Amazon JAVA SDKs for all the services I was using: (SQS, EC2 and S3). To ensure that the code is modular and readable, I separated the repository from the services and controller. Based on the project requirement, I configured a POST URL for receiving the image from the user, implemented logic for sending messages to the queue, and consequently listening to the output queue for the result. For the next part, I dealt with scaling up the application. Based on the design, I implemented a hash table and cached the output for a quick response to the user.

Testing

To make sure that the application was bug free, I tested the application extensively. The first step was to test the part for which I was responsible (web tier). For web tier, I tested the following:

- Correct number of app tier instances spawned for different numbers of messages in the Input Queue. (tested with 10,20,30 ..100 messages in Input Queue)
- The number of messages in the Input Queue for a different number of POST Requests for sending images to the server.
- The number of messages in the Output Queue after the app successfully identified the image.

After testing out the individual tiers, I tested the application upon integration. Tests were performed from both REST API Clients(Insomnia, Postman) and client. I also did a lot of regression and made sure the app behaved correctly after multiple code changes. On the final version of the application, I did scalability testing for upto 1000 messages for the application and found out that it was working correctly.

Individual contributions

Deepakshi Mittal

This project was a great learning experience for me. I learned how to use AWS EC2, S3, SQS services to build a scalable application. I also learned a lot about servers, REST APIs, networking, message communication protocols, file transfer protocols..

Design

Design phase involved figuring out how to use the Amazon web Services, how one module will communicate with the other, where each component will reside in the proposed architecture, deciding the request, response format of each component. I collaborated with my team members to plan the High Level design of the application, how to effectively design the app tier so that we can test individual modules and integrate all later. I also brainstormed with my team the technologies to be used for each component, including frontend and backend. My main contribution in terms of code was the app tier.

Implementation

I implemented the App tier using AWS Java SDK, Spring Boot, Maven. The input for running the APP tier is fully configurable and is taken through Command line specifying the Request Queue, Response Queue, Input Bucket and Output bucket. I used method as much as possible to make code Modular, reusable and easy to understand. I wrote the functionality for reading objects from S3 bucket and putting response object to S3 bucket output bucket, terminating, stopping the instances. I suggested that we use a Amazon Machine Image(AMI) for the App tier for creating the new EC2 Instances. It contains the application jar and the deep learning model.

Testing

I unit tested the App tier using Command Line with different parameters. I did extensive testing of all components after the web and app tier were integrated. I also tested the application end to end for single requests through Postman. I also made sure the application works fine with the provided workload generator, and is able to handle concurrent requests with efficiency. The categorization results from the app tier's deep learning model were compared to the expected output. The predicted output was compared to the output queue messages and output bucket result. Finally, the auto scale-in feature was tested by seeing if the instances terminated when no more requests were waiting to be served.
Following the integration, the complete project was tested against 1 to 1000 requests using the provided workload generator to check that the categorization results matched the expected results. Finally, the AWS Console was utilized to test the scaling out and scaling in of EC2 instances. SQS queues were watched to verify that messages were stored, retrieved, and discarded successfully. S3 was double-checked to make sure the input and classification results were stored properly.

Individual contributions

Karan Ganeshdas Barhanpur

Design:

The app-logic component was created in such a way that it could only handle one request from the request queue at a time. Once the request has been received, the categorization result will be loaded into the deep learning model. The results of the following categorization would be saved in the response queue and shown in the app-frontend. These findings would be saved in an S3 output bucket for long-term storage. After catering a request, the app-logic would wait for a certain amount of time and then check the request queue again to see if there were any additional requests. If the answer is yes, the procedure will proceed in the same manner. If there were no more requests in the queue, the app-logic would self-terminate, resulting in the application being scaled down. The notion of putting the scale-in logic in the app-logic component came to me since it would result in a better design.

Implementation

As an AWS Java project, I built the aforementioned app-logic design. This component's input is provided by command line parameters, and values are derived by parsing them. There will be no hard coding in the application, hence this is a superior design implementation. The functionality to take in data, process it, save it, and self-terminate were all fine-grained to particular methods that would do their job. I used the idea of threads to impose pauses between iterations as well as wait for the instance to self-terminate. Several repetitive processes were recognized and converted into methods that could be utilized across the program. The app-logic can withstand a variety of faults. Various access modifiers and exception handling methods, such as try-catch blocks, were used to accomplish this.

Testing

Each of the app-logic component's individual methods was unit tested to ensure that they functioned properly. All of the methods were then combined in the following phase to verify the overall functionality of the app-logic. In order to do this testing, dummy items in the request queue were created. To ensure that the app-logic was providing the needed responses, the S3 output bucket and response queue were watched. By producing the jar file and running it with command line options, the major functionality testing of app-logic was carried out in local and EC2 instances. Finally, this component was connected with the rest of the project, and the app-logic EC2 instances were watched from the AWS interface to ensure that the auto scale feature was enabled when the number of requests in the queue decreased or became empty. With the use of the AWS console and app-frontend, the S3 output bucket and SQS response queue were also evaluated to see if they delivered the expected results. By thoroughly testing and analyzing the application's behavior, I was able to estimate the time range for self-termination.