CSE 546 — Project Report

Samarth Patel (1220111230)

Shubham Bansal (1224102750)

Sulabh Soneji (1219617139)

1. Problem statement

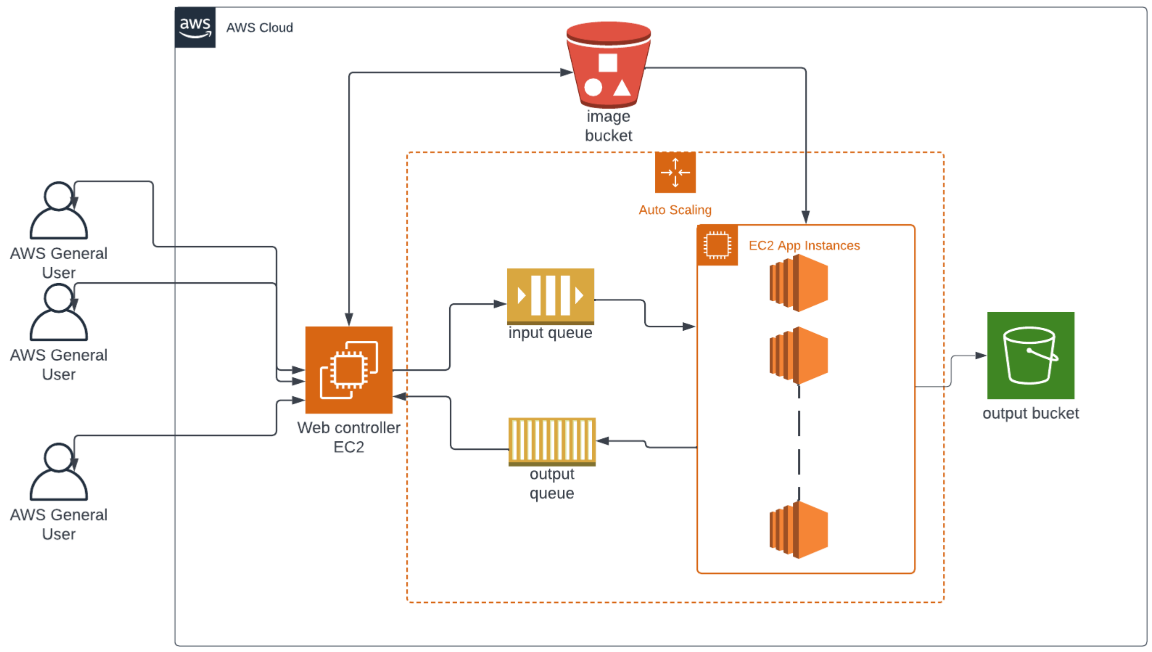
The project aims at creating an elastic application that will help to recognize images using deep learning models and IaaS resources from AWS. The application will be REST service exposed to the clients in terms of Image recognition to predict what kind of image it is. Our application will take an image as an input to the REST api and will return output in the form of plain text which would be the prediction result from the deep learning model. We are using AWS for communication, computation and storage. Summing up it involves a list of tasks like: developing a Rest API, architecture design, creating a load balancer that would scale-in and scale-out based on requirements, and testing of the whole application.

2. Design and implementation

2.1 Architecture

AWS Services:

- **AWS EC2:**
  - EC2 is being used to run the application on a virtual machine in AWS. For the project, we used EC2 Instances in order to implement the project at the Web Controllers and the App Instances.
- **AWS S3 bucket:**
  - S3 buckets are used for storing the input images and results of the image recognition corresponding to those images. The users will request images to be recognized using Web controllers, and the output would be received from the App Instances, once the classification is completed. The output format would be (key, value) pair, where the key would be imageId and the value would be the prediction.
- **AWS SQS:**
  - SQS Queues are used as input and output queues. Here the input queues store the requests from the Web Controller, and the output queue stores output from the App Instances. So the queues are used to connect the Web controller and App Instances.



Architecture Description:

For the architecture design, we have one EC2 instance in Web Controller, which takes multiple requests from the users. After taking all those requests the EC2 instance feeds the input queue with all the requests and then those requests are fed to the App Instances(1 minimum, 19 maximum), furthermore all of these requests are stored in the S3 buckets. The single Ec2 instance in the Web Controller helps to scale-in and scale-out using the load balancing algorithm which decides to scale out based on the EC2 instances and the demand in the SQS Queues. It can maximum scale out to 20 app-instances as that is the limit for the free tier. We have designed the blocking hashmap dictionary structure which handles the concurrency issue in a better way. The web-tier instance also works as a controller for receiving and handling requests to a certain http endpoint.

The App Instances run a deep learning algorithm and predict the output for user requested images, by taking requests from the input queue. App instances automatically scales-in according to the demand. Scaling in is handled by app instances themselves as they will be terminated when they are unable to find any new input requests in the SQS input queue. After the prediction is completed the results are stored in S3 bucket in (key, value) form. And the results are feeded into the output queue.

Our architecture is also capable of handling the fault tolerance on App Tier. We have used the visibility timeout feature of SQS. It is responsible for making the message invisible to other EC2 app instances when one EC2 app instance is already reading the message. Considering the maximum amount of time which the given deep learning algorithm takes to predict the result, we have set the visibility timeout to 100 seconds. In an imaginary scenario, if something goes wrong with the current working EC2 app instance, the message will be visible to other app instances after those 100 seconds, otherwise it deletes the message from the input queue after the processing of the message is complete and stores the result in the S3 bucket and feeds the message in the output queue.

2.2 Autoscaling

Autoscaling will be done by the web controller instance and app instances. The web controller stores new incoming requests in the input queue as shown in the architecture diagram. The number of messages queued in the SQS can be used to scale out as per demand.

For scaling-out, we are considering the number of messages in the input queue and the total number of EC2 instances running. We balanced the EC2 instances based on the number of request messages. If the number of messages is greater than the number of running EC2 app instances, we will increase the number of EC2 app instances according to the free tier EC2 instances available. We had to make sure that new app instances should not exceed the number of available instances in the free tier, so we have stored the instructed number of max instances in a constant and we will run the new required instances accordingly. The above-mentioned process will be continuously responsible for scaling out and will be executed in a while loop.

Scaling in will be handled through the app-tier instances. When an app instance successfully predicts the image, it puts the output in output SQS and also stores the output in the output S3 bucket as key-value pairs. The app instance will perform the above-given procedure until there are messages available in the input queue, If it cannot find any request messages in the queue, it will terminate itself. All the app instances will terminate automatically when there are no messages in the SQS. The app instances will have a wait time of 100 milliseconds after delivering the first output so that it will allow the SQS enough time to poll any other available messages. That way there won't be any idle app-instance running and hence fulfilling the scaling-in requirement.

2.3 Member Tasks

Samarth Patel

Design Contributions:-

1. Collaborated with team members and brainstormed about the possible design of the project. We came up with two different designs, one for sequential execution of

app-instances and one which handles the task parallely. Drafted the final design with Shubham and Sulabh, decided to go with Java Spring Boot framework.
2. Brainstormed the possibilities for scaling out the app-tier according to the number of messages in input queue.
3. Decided to use the blocking hashmap data structure to better handle the concurrency of input and output messages.

Code Contributions:-

1. Coded the logic for blocking hashmap in ImageRecognitionServiceImpl.java file of the aws-web-service module.
2. Created a java class which holds the value of the aws configure contents throughout the application in aws-web-service, aws-app-service, and aws-web-service-running modules.
3. Helped and collaborated with Shubham to build the runDeepLearningAlgorithm method of the DeepLearningServiceImpl.java class which manages the creation and execution of deep learning algorithm from the given disk image.
4. receiveOutput method in the LoadBalancerServiceImpl.java in the aws-web-service module, which is responsible for fetching the output queue and poll all the messages available in the output SQS, process the message and finally delete the processed message from the input queue.
5. Helped Shubham with the method recognizeImage in DeepLearningServiceImpl class which is responsible for processing the request input messages and providing the top-1 result from the deep learning algorithm.

Testing Contributions:-

1. Created the jar files and copied it to the instances in cloud, ensured the fault-less creation of the app-instances with the command.
2. Monitored the working of the EC2 instances while scaling out and scaling in as per the demand, and made sure it is working properly.
3. Performed the final round of testing, tested the whole working of the project from command line, ensure the generated output is as intended. Tested the creation, deletion of the ec2 instances, file upload/deletion from S3 buckets, and receiving and deletion of the messages from SQS.

Project Report Contributions:-

1. Designed and draw the architectural diagram for the project.
2. Explained the working and structure of the cloud architecture.
3. Elaborated the working of the different app-tier and web-tier modules of the project.
4. Collaborated with Shubham and finalized the setup and execution of the project.

Sulabh Soneji

Design Contributions:-

1. Brainstormed the ideas for the architecture and how all the components will be linked using AWS. The important task was to incorporate the Web controller instance and the App Instances.
2. Helped to finalize the architecture and the flow of the design.

Code Contributions:-

1. AppConfig class which will be used for the creation of the AWS EC2, AWS SQS, and AWS S3, and Load balancer for service.
2. EC2RepositoryImpl for all the operations related to the instance. Those operations mainly involve creating instances, starting instances, stopping instances and terminating instances, and listing of all the EC2 instances.
3. Find the number of running instances among the App Instances.
4. Worked on the scale in and scale out of the application using the LoadBalancer service class.
5. S3RepositoryImpl for creating buckets, deleting buckets and uploading the file to the buckets.
6. SQSRepositoryImpl was implemented for the creation of queue, deletion of queue, sending messages, receiving messages.

Testing Contributions:-

1. As a team we tested the whole functionality of the application. We checked the flow by giving different images and checked if the messages sent through the SQS Queues were perfect or not, and verified the predicted outputs from the App Instances.
2. Tested all the basic functionalities like creation of the EC2 Instances, S3 Buckets, and SQS Queues.
3. Also checked and verified the Queues by sending messages, EC2 instances were verified by checking if they were running or not, and S3 buckets were verified by uploading the basic files.

Project Report Contributions:-

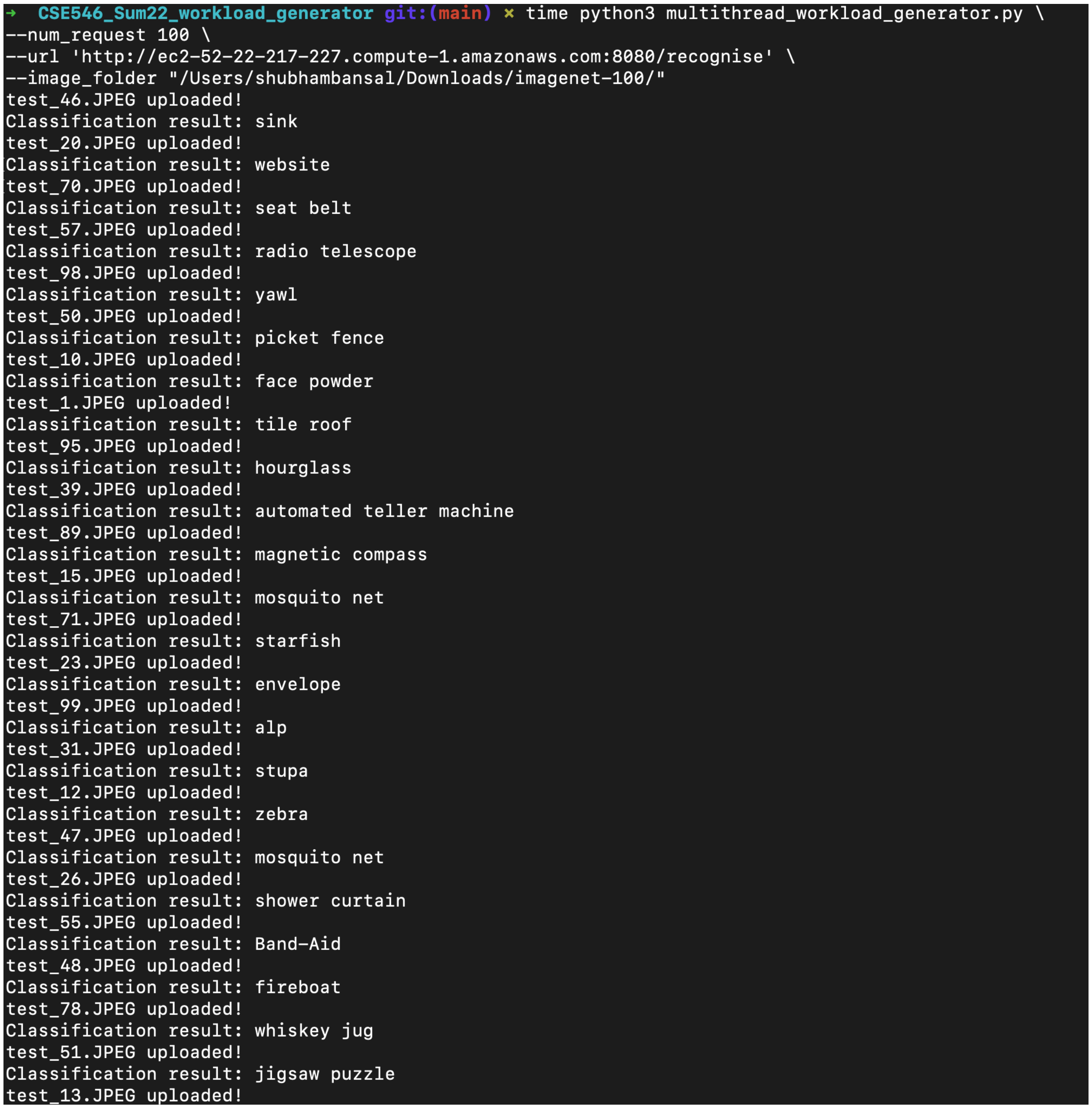
1. Helped the team with writing the project statement.
2. Completed the description and the use of all the AWS Services in the project.
3. Also explained in detail about how the architecture works in the project.
4. Furthermore, added all the details about the testing steps.
5. Atlast, added all the details and the testing results that have been performed by the team

3. Testing and evaluation

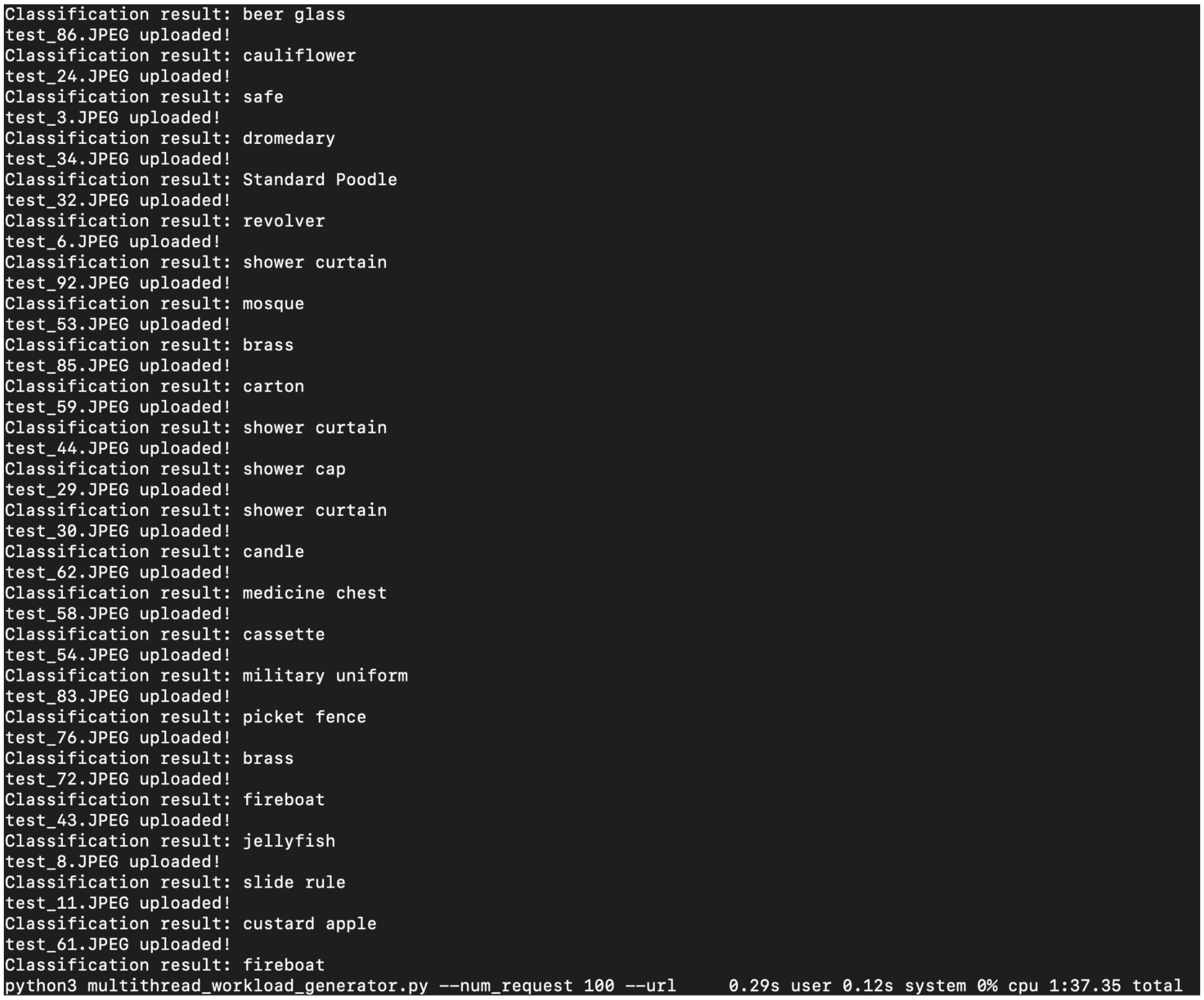
- **Steps taken for testing the application:**
    - Testing our application for the implementation of auto-scaling and load balancing by checking the number of running instances on the EC2 dashboard.
    - Check the number of running app instances in order to not exceed the limit of 20 and also submit all the pending requests to the queue.
    - Checking how much time it takes to complete all the requests (it was less than 2 minutes).
    - Added logs every time to make sure that all the required operations are running and executed in the required sequence.
    - Furthermore, the dataset given was tested and the outputs predicted they did match with the corresponding input.

- **Testing and evaluation results:**

Testing results for uploading and recognising the image

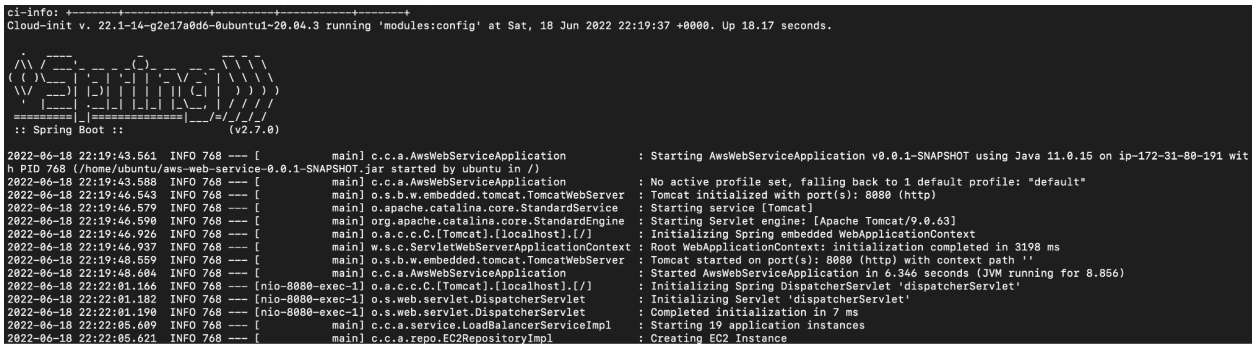


```
➜  CSE546_Sum22_workload_generator git:(main) ✗ time python3 multithread_workload_generator.py \
--num_request 100 \
--url 'http://ec2-52-22-217-227.compute-1.amazonaws.com:8080/recognise' \
--image_folder "/Users/shubhambansal/Downloads/imagenet-100/"
test_46.JPEG uploaded!
Classification result: sink
test_20.JPEG uploaded!
Classification result: website
test_70.JPEG uploaded!
Classification result: seat belt
test_57.JPEG uploaded!
Classification result: radio telescope
test_98.JPEG uploaded!
Classification result: yawl
test_50.JPEG uploaded!
Classification result: picket fence
test_10.JPEG uploaded!
Classification result: face powder
test_1.JPEG uploaded!
Classification result: tile roof
test_95.JPEG uploaded!
Classification result: hourglass
test_39.JPEG uploaded!
Classification result: automated teller machine
test_89.JPEG uploaded!
Classification result: magnetic compass
test_15.JPEG uploaded!
Classification result: mosquito net
test_71.JPEG uploaded!
Classification result: starfish
test_23.JPEG uploaded!
Classification result: envelope
test_99.JPEG uploaded!
Classification result: alp
test_31.JPEG uploaded!
Classification result: stupa
test_12.JPEG uploaded!
Classification result: zebra
test_47.JPEG uploaded!
Classification result: mosquito net
test_26.JPEG uploaded!
Classification result: shower curtain
test_55.JPEG uploaded!
Classification result: Band-Aid
test_48.JPEG uploaded!
Classification result: fireboat
test_78.JPEG uploaded!
Classification result: whiskey jug
test_51.JPEG uploaded!
Classification result: jigsaw puzzle
test_13.JPEG uploaded!
```

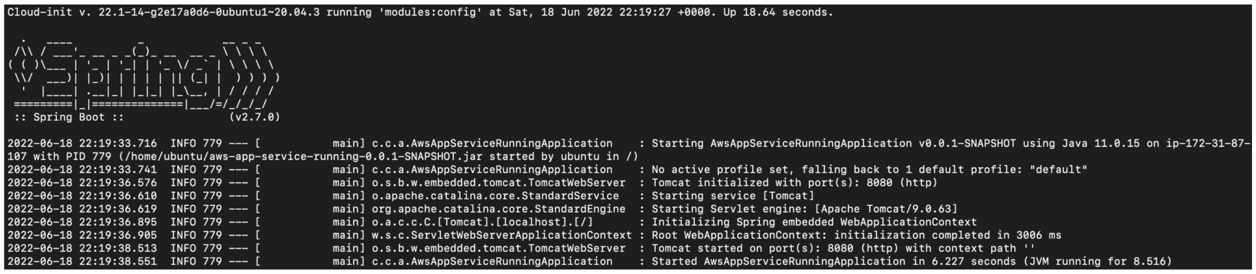


```
Classification result: beer glass
test_86.JPEG uploaded!
Classification result: cauliflower
test_24.JPEG uploaded!
Classification result: safe
test_3.JPEG uploaded!
Classification result: dromedary
test_34.JPEG uploaded!
Classification result: Standard Poodle
test_32.JPEG uploaded!
Classification result: revolver
test_6.JPEG uploaded!
Classification result: shower curtain
test_92.JPEG uploaded!
Classification result: mosque
test_53.JPEG uploaded!
Classification result: brass
test_85.JPEG uploaded!
Classification result: carton
test_59.JPEG uploaded!
Classification result: shower curtain
test_44.JPEG uploaded!
Classification result: shower cap
test_29.JPEG uploaded!
Classification result: shower curtain
test_30.JPEG uploaded!
Classification result: candle
test_62.JPEG uploaded!
Classification result: medicine chest
test_58.JPEG uploaded!
Classification result: cassette
test_54.JPEG uploaded!
Classification result: military uniform
test_83.JPEG uploaded!
Classification result: picket fence
test_76.JPEG uploaded!
Classification result: brass
test_72.JPEG uploaded!
Classification result: fireboat
test_43.JPEG uploaded!
Classification result: jellyfish
test_8.JPEG uploaded!
Classification result: slide rule
test_11.JPEG uploaded!
Classification result: custard apple
test_61.JPEG uploaded!
Classification result: fireboat
python3 multithread_workload_generator.py --num_request 100 --url     0.29s user 0.12s system 0% cpu 1:37.35 total
```

Creating the instances and starting the web services application

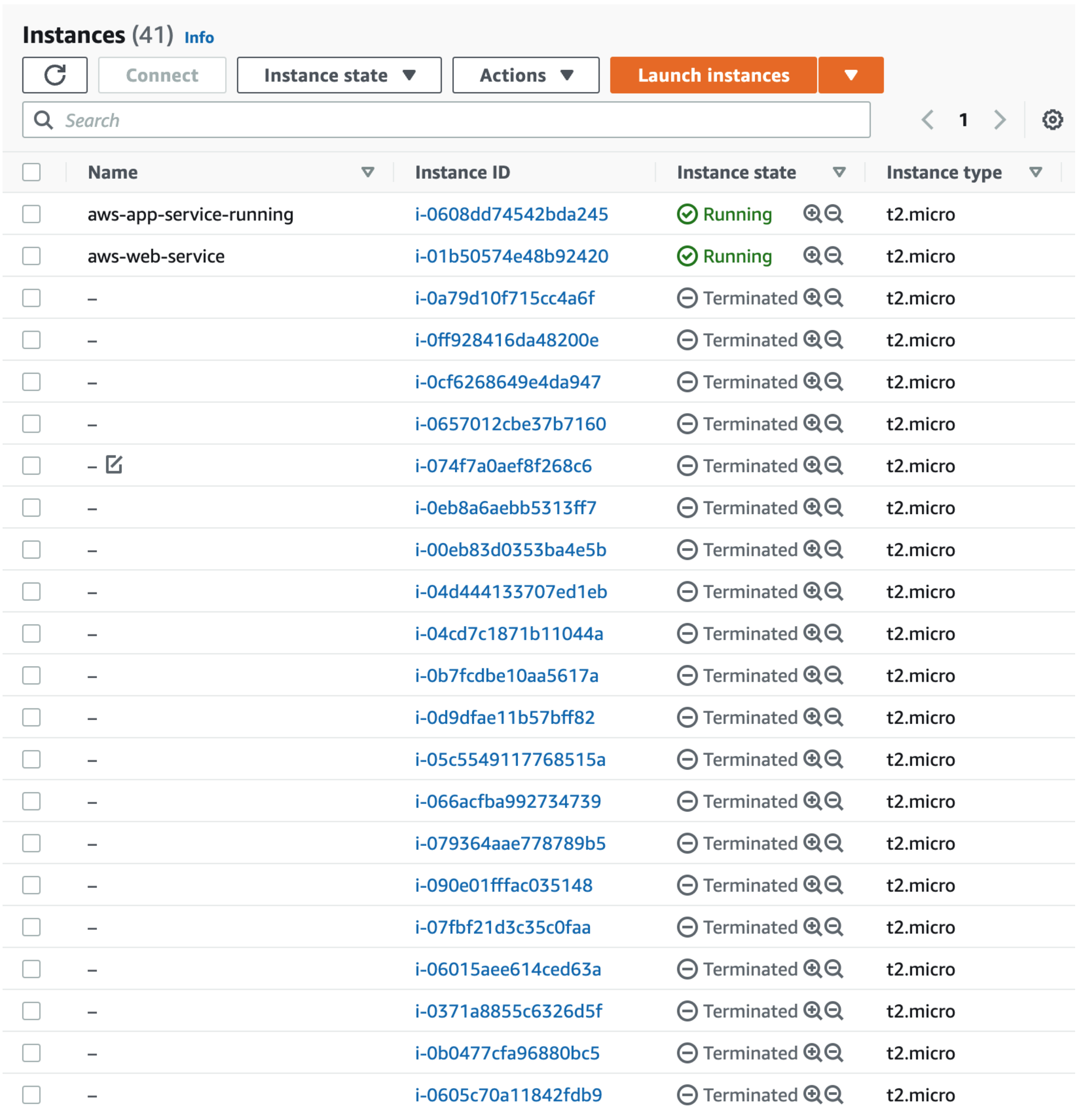


```
ci-info: +--------+-------------+---------+----------+-------+
Cloud-init v. 22.1-14-g2e17a0d6-0ubuntu1~20.04.3 running 'modules:config' at Sat, 18 Jun 2022 22:19:37 +0000. Up 18.17 seconds.

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.7.0)

2022-06-18 22:19:43.561  INFO 768 --- [           main] c.c.a.AwsWebServiceApplication           : Starting AwsWebServiceApplication v0.0.1-SNAPSHOT using Java 11.0.15 on ip-172-31-80-191 wit
h PID 768 (/home/ubuntu/aws-web-service-0.0.1-SNAPSHOT.jar started by ubuntu in /)
2022-06-18 22:19:43.588  INFO 768 --- [           main] c.c.a.AwsWebServiceApplication           : No active profile set, falling back to 1 default profile: "default"
2022-06-18 22:19:46.543  INFO 768 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2022-06-18 22:19:46.579  INFO 768 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2022-06-18 22:19:46.590  INFO 768 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.63]
2022-06-18 22:19:46.926  INFO 768 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2022-06-18 22:19:46.937  INFO 768 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 3198 ms
2022-06-18 22:19:48.559  INFO 768 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2022-06-18 22:19:48.604  INFO 768 --- [           main] c.c.a.AwsWebServiceApplication           : Started AwsWebServiceApplication in 6.346 seconds (JVM running for 8.856)
2022-06-18 22:22:01.166  INFO 768 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2022-06-18 22:22:01.182  INFO 768 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2022-06-18 22:22:01.190  INFO 768 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 7 ms
2022-06-18 22:22:05.609  INFO 768 --- [           main] c.c.a.service.LoadBalancerServiceImpl    : Starting 19 application instances
2022-06-18 22:22:05.621  INFO 768 --- [           main] c.c.a.repo.EC2RepositoryImpl             : Creating EC2 Instance
```



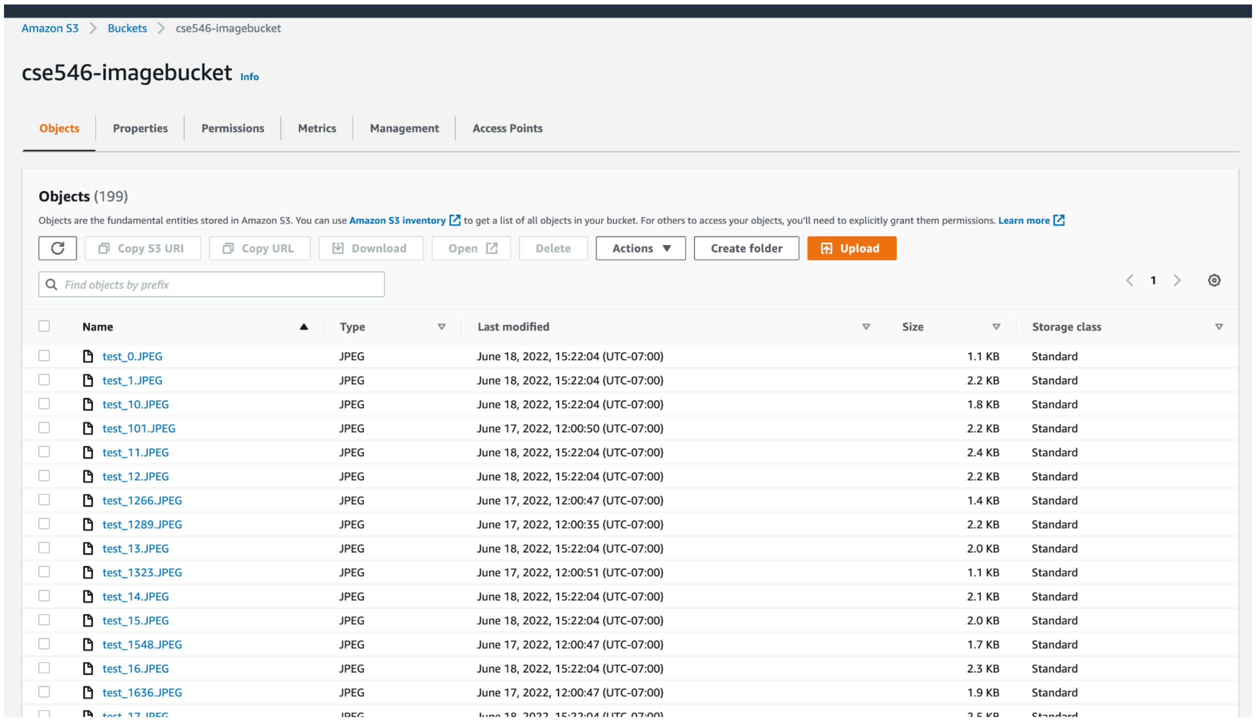
```
Cloud-init v. 22.1-14-g2e17a0d6-0ubuntu1~20.04.3 running 'modules:config' at Sat, 18 Jun 2022 22:19:27 +0000. Up 18.64 seconds.

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.7.0)

2022-06-18 22:19:33.716  INFO 779 --- [           main] c.c.a.AwsAppServiceRunningApplication    : Starting AwsAppServiceRunningApplication v0.0.1-SNAPSHOT using Java 11.0.15 on ip-172-31-87-
107 with PID 779 (/home/ubuntu/aws-app-service-running-0.0.1-SNAPSHOT.jar started by ubuntu in /)
2022-06-18 22:19:33.741  INFO 779 --- [           main] c.c.a.AwsAppServiceRunningApplication    : No active profile set, falling back to 1 default profile: "default"
2022-06-18 22:19:36.576  INFO 779 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2022-06-18 22:19:36.610  INFO 779 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2022-06-18 22:19:36.619  INFO 779 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.63]
2022-06-18 22:19:36.895  INFO 779 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2022-06-18 22:19:36.905  INFO 779 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 3006 ms
2022-06-18 22:19:38.513  INFO 779 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2022-06-18 22:19:38.551  INFO 779 --- [           main] c.c.a.AwsAppServiceRunningApplication    : Started AwsAppServiceRunningApplication in 6.227 seconds (JVM running for 8.516)
```

Check that the instances were created and terminated after the use



Name	Instance ID	Instance state	Instance type
aws-app-service-running	i-0608dd74542bda245	Running	t2.micro
aws-web-service	i-01b50574e48b92420	Running	t2.micro
–	i-0a79d10f715cc4a6f	Terminated	t2.micro
–	i-0ff928416da48200e	Terminated	t2.micro
–	i-0cf6268649e4da947	Terminated	t2.micro
–	i-0657012cbe37b7160	Terminated	t2.micro
–	i-074f7a0aef8f268c6	Terminated	t2.micro
–	i-0eb8a6aebb5313ff7	Terminated	t2.micro
–	i-00eb83d0353ba4e5b	Terminated	t2.micro
–	i-04d444133707ed1eb	Terminated	t2.micro
–	i-04cd7c1871b11044a	Terminated	t2.micro
–	i-0b7fcdbe10aa5617a	Terminated	t2.micro
–	i-0d9dfae11b57bff82	Terminated	t2.micro
–	i-05c5549117768515a	Terminated	t2.micro
–	i-066acfba992734739	Terminated	t2.micro
–	i-079364aae778789b5	Terminated	t2.micro
–	i-090e01fffac035148	Terminated	t2.micro
–	i-07fbf21d3c35c0faa	Terminated	t2.micro
–	i-06015aee614ced63a	Terminated	t2.micro
–	i-0371a8855c6326d5f	Terminated	t2.micro
–	i-0b0477cfa96880bc5	Terminated	t2.micro
–	i-0605c70a11842fdb9	Terminated	t2.micro

Images have been added to the Input bucket



Amazon S3 > Buckets > cse546-imagebucket

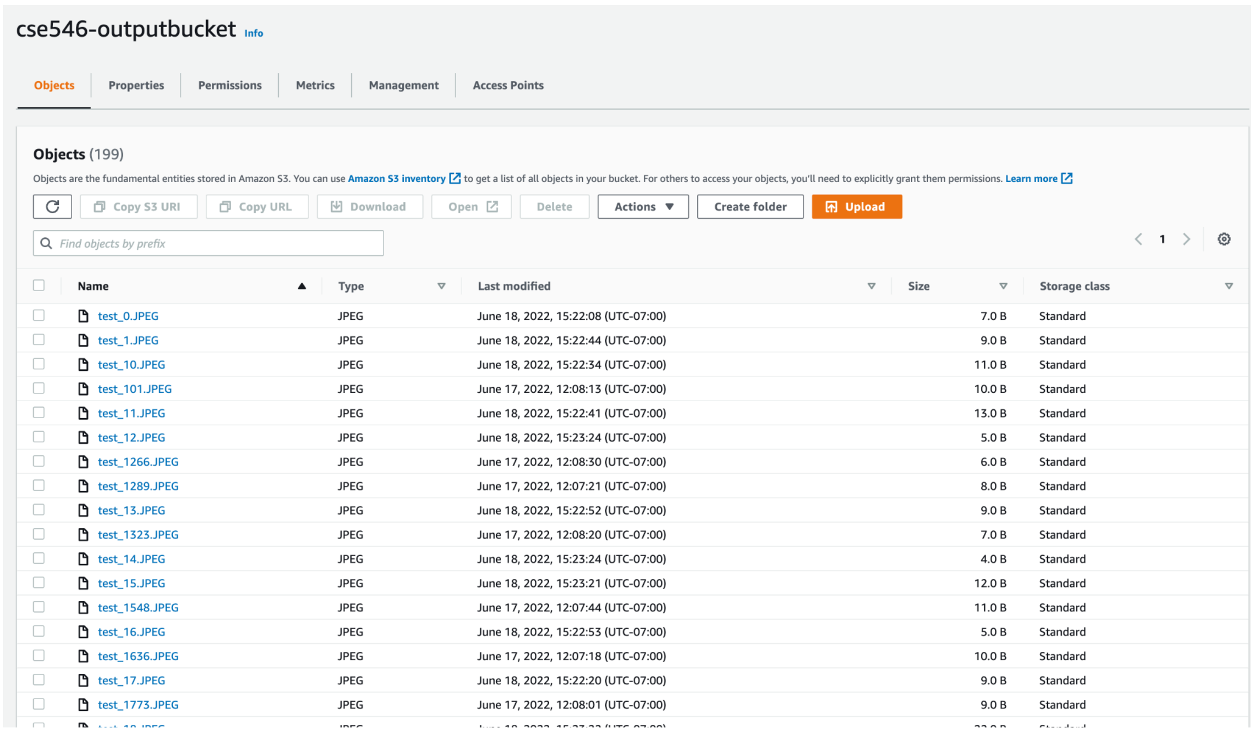
cse546-imagebucket Info

Objects (199)

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_0.JPEG	JPEG	June 18, 2022, 15:22:04 (UTC-07:00)	1.1 KB	Standard
test_1.JPEG	JPEG	June 18, 2022, 15:22:04 (UTC-07:00)	2.2 KB	Standard
test_10.JPEG	JPEG	June 18, 2022, 15:22:04 (UTC-07:00)	1.8 KB	Standard
test_101.JPEG	JPEG	June 17, 2022, 12:00:50 (UTC-07:00)	2.2 KB	Standard
test_11.JPEG	JPEG	June 18, 2022, 15:22:04 (UTC-07:00)	2.4 KB	Standard
test_12.JPEG	JPEG	June 18, 2022, 15:22:04 (UTC-07:00)	2.2 KB	Standard
test_1266.JPEG	JPEG	June 17, 2022, 12:00:47 (UTC-07:00)	1.4 KB	Standard
test_1289.JPEG	JPEG	June 17, 2022, 12:00:35 (UTC-07:00)	2.2 KB	Standard
test_13.JPEG	JPEG	June 18, 2022, 15:22:04 (UTC-07:00)	2.0 KB	Standard
test_1323.JPEG	JPEG	June 17, 2022, 12:00:51 (UTC-07:00)	1.1 KB	Standard
test_14.JPEG	JPEG	June 18, 2022, 15:22:04 (UTC-07:00)	2.1 KB	Standard
test_15.JPEG	JPEG	June 18, 2022, 15:22:04 (UTC-07:00)	2.0 KB	Standard
test_1548.JPEG	JPEG	June 17, 2022, 12:00:47 (UTC-07:00)	1.7 KB	Standard
test_16.JPEG	JPEG	June 18, 2022, 15:22:04 (UTC-07:00)	2.3 KB	Standard
test_1636.JPEG	JPEG	June 17, 2022, 12:00:47 (UTC-07:00)	1.9 KB	Standard

Results have been added to the Ouput bucket



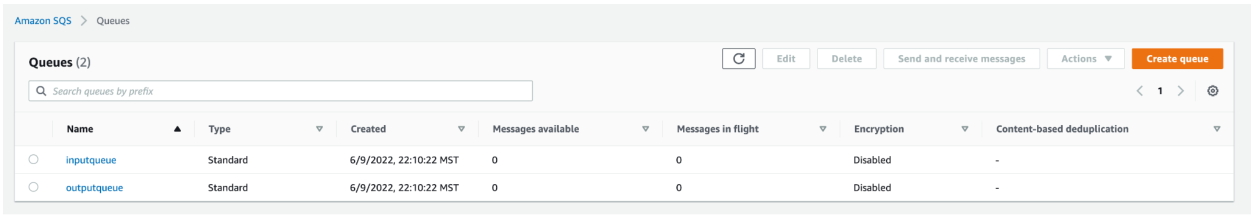
cse546-outputbucket Info

Objects (199)

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_0.JPEG	JPEG	June 18, 2022, 15:22:08 (UTC-07:00)	7.0 B	Standard
test_1.JPEG	JPEG	June 18, 2022, 15:22:44 (UTC-07:00)	9.0 B	Standard
test_10.JPEG	JPEG	June 18, 2022, 15:22:34 (UTC-07:00)	11.0 B	Standard
test_101.JPEG	JPEG	June 17, 2022, 12:08:13 (UTC-07:00)	10.0 B	Standard
test_11.JPEG	JPEG	June 18, 2022, 15:22:41 (UTC-07:00)	13.0 B	Standard
test_12.JPEG	JPEG	June 18, 2022, 15:23:24 (UTC-07:00)	5.0 B	Standard
test_1266.JPEG	JPEG	June 17, 2022, 12:08:30 (UTC-07:00)	6.0 B	Standard
test_1289.JPEG	JPEG	June 17, 2022, 12:07:21 (UTC-07:00)	8.0 B	Standard
test_13.JPEG	JPEG	June 18, 2022, 15:22:52 (UTC-07:00)	9.0 B	Standard
test_1323.JPEG	JPEG	June 17, 2022, 12:08:20 (UTC-07:00)	7.0 B	Standard
test_14.JPEG	JPEG	June 18, 2022, 15:23:24 (UTC-07:00)	4.0 B	Standard
test_15.JPEG	JPEG	June 18, 2022, 15:23:21 (UTC-07:00)	12.0 B	Standard
test_1548.JPEG	JPEG	June 17, 2022, 12:07:44 (UTC-07:00)	11.0 B	Standard
test_16.JPEG	JPEG	June 18, 2022, 15:22:53 (UTC-07:00)	5.0 B	Standard
test_1636.JPEG	JPEG	June 17, 2022, 12:07:18 (UTC-07:00)	10.0 B	Standard
test_17.JPEG	JPEG	June 18, 2022, 15:22:20 (UTC-07:00)	9.0 B	Standard
test_1773.JPEG	JPEG	June 17, 2022, 12:08:01 (UTC-07:00)	9.0 B	Standard

Queues are created and they are ready for sending and receiving messages


Amazon SQS > Queues

Queues (2)

Name	Type	Created	Messages available	Messages in flight	Encryption	Content-based deduplication
inputqueue	Standard	6/9/2022, 22:10:22 MST	0	0	Disabled	-
outputqueue	Standard	6/9/2022, 22:10:22 MST	0	0	Disabled	-

4. Code

The code for the given project is developed in Java Spring Boot framework. The code is divided into 2 modules according to the requirements of the web-tier and app-tier. The aws-app-service-running is an alternative to aws-app-service which doesn't shutdown automatically to tackle the sequential API request case which will help with removing the startup and shutdown time of the vm for each API calls.

1. Aws-web-service
    a. A rest controller **ServiceController.java:**
        - This controller maps the incoming image requests to the function recognizeImage in ImageRecognitionService.java file.
        - The recognizeImage function sends the name of the image in input SQS and also uploads the image received from the to input S3 bucket.
        - We have used the blocking hashmap data structure to overcome possible concurrency issues.

    b. A service function *receiveOutput* in **ImageRecognitionService.java:**
        - This service is executed in a different thread which is responsible for receiving messages from the output SQS, putting the output in the blocking hashmap and deleting the message from the output queue after it is successfully processed.

    c. A service function *scaleInOrOut* in **LoadBalancerServiceImpl.java:**
        - This service is responsible for scaling in the computational instances as per the available messages in the input queue.
        - The detailed working can be seen in the scaling section of auto balancing.

2. Aws-app-service

    a. A deep learning service for recognise images in **DeepLearningServiceImpl.java:**

        - This service is run by individual app instances mentioned in the architecture diagram.
        - It will receive the messages from the input queue, and if no messages are available then it will terminate the instance (not for 'aws-app-service-running'),hence providing scaling out in requirement.
        - If there is a message available in the input SQS, then it will download the image locally in the /tmp/ folder and provide the path of the image to the classification deep learning module and after that cleanup the image from the /tmp/ folder.
        - It will build a process to execute the deep learning classification algorithm for the image and the result of the classification will be stored in the output S3 bucket.
        - Also, the output will be fed to the output SQS and that message will be finally be deleted from the input SQS.

3. Aws-app-service-running (Alternative app-tier instance logic)

    a. As explained in the details of the architecture, we have also developed the sequential executer of the requests in this type of module. However, this is best suitable for sequential API calls to reduce the vm creation and termination time.
    b. This module keeps an app-tier EC2 instance always running and it only solely processes the image recognition task reading the message from the input queue. After processing the one image (or one request message), it will sleep for 100 milliseconds and then it will try to fetch another message from the input queue.
    c. The code for above-explained working is in **DeepLearningServiceImpl.java** file in the aws-app-service-running module. The only difference between this module and module aws-app-service is that, this module will not shutdown if the messages are not present in the SQS queue.

Steps to install and run the program

- Setup of Web Tier Instance (aws-web-service)
    - Create a jar file using "mvn package".
    - Create a new instance of the from the AMI image "ami-0bb1040fdb5a076bc"
    - Create the java environment by installing the jdk available in the instance: sudo apt install default-jdk
    - Copy the build jar file to the IAAS VM instance using scp : "scp -i shubham.bansal.pem aws-web-service-0.0.1-SNAPSHOT.jar ubuntu@ec2-52-22-217-227.compute-1.amazonaws.com:/home/ubuntu/"
    - Start the service using command "sudo -u ubuntu java -jar aws-web-service-0.0.1-SNAPSHOT.jar" in home folder of ubuntu user.

- Setup of App Tier Instance (aws-app-service-running)

    - Create a jar file using “mvn package”.
    - Create a new instance of the from the AMI image “ami-0bb1040fdb5a076bc”
    - Create the java environment by installing the jdk available in the instance: sudo apt install default-jdk
    - Copy the build jar file to the IAAS VM instance using scp :
      “scp -i shubham.bansal.pem aws-app-service-running-0.0.1-SNAPSHOT.jar ubuntu@<vm-url>:/home/ubuntu/”
    - Create “/var/lib/cloud/scripts/per-boot/aws-app-service-running.sh” and add following
      “sudo -u ubuntu java -jar aws-app-service-running-0.0.1-SNAPSHOT.jar”
    - Execute “sudo chmod +x /var/lib/cloud/scripts/per-boot/aws-app-service-running.sh” to give the execution permission to the bash file.
    - Bash script “/var/lib/cloud/scripts/per-boot/aws-app-service-running.sh” will run automatically when the vm starts and will execute the service jar file.

- Setup of Dynamic App Tier instance (aws-app-service)
    - This service is almost same as aws-app-service-running and setup the same way, but does shuts down if there are no input message in the SQS inputqueue.
    - This helps with auto scaling in/out of the app service instances.

Individual Contribution Report

Shubham Bansal (1224102750)

Design Contributions:-

1. Designed the architecture of the project including all three services “aws-web-service”, “aws-app-service”, and “aws-app-service-running”.
2. Discussed the service framework to use for building different services within the time constraint that we had.
3. Managed the project deliverables, code quality, and team members.

Code Contributions:-

6. Service: aws-web-service
    a. ServiceController class which is the controller class for the frontend web API for image recognition.
    b. SQSRepositoryImpl class which is the implementation of the SQSRepository interface to have SQS functionality available to our service.
    c. ImageRecognitionServiceImpl class which is the implementation of the ImageRecognitionService interface and handles the communication with the SQS queue for input and output messaging
    d. LoadBalancerServiceImpl class which is the implementation of the LoadBalancerService interface and handles the scaling in and out of the “aws-app-service” service.
7. Service: aws-app-service
    a. AwsConfig class which is a SpringBoot configuration class that initiates AWS SQS client, and AWS S3 client for service to use.
    b. SQSRepositoryImpl class which is the implementation of the SQSRepository interface to have SQS functionality available to our service.
    c. Method recognizeImage in DeepLearningServiceImpl class which is the implementation of the DeepLearningService interface.
8. Service: aws-app-service-running
    a. AwsConfig class which is a SpringBoot configuration class that initiates AWS SQS client, and AWS S3 client for service to use.
    b. SQSRepositoryImpl class which is the implementation of the SQSRepository interface to have SQS functionality available to our service.
    c. Method recognizeImage in DeepLearningServiceImpl class which is the implementation of the DeepLearningService interface.

Testing Contributions:-

1. The first round of testing on AWS with setting up the input S3 bucket ‘cse546-imagebucket’ and output bucket ‘cse546-outputbucket’.
2. Also, setting up the ‘aws-app-service’, ‘aws-app-service-running’, and ‘aws-web-service’ for the first time. Providing those setup instructions to the rest of the team for final rounds of testing.
3. Finally verified the service output in the output bucket ‘cse546-outputbucket’.

Project Report Contributions:-

1. Helped with writing design and architecture documentation in the project report.
2. Also, I wrote down the testing setup instructions and execution setup instructions for the project report.
3. Provided the screenshot of the live testing of the service to be included in the project report.