CSE 546 - Project Report

Group members:

S.NO	Name	Mail ID	Student ID
1	Avinash Kodali	akodali5@asu.edu	1224993120
2	Manogna Pagadala	mpagada1@asu.edu	1224798575
3	Manaswi Kommini	mkommini@asu.edu	1226342288

1. Problem statement

In this project, we are creating an adaptable image recognition software that recognizes user-provided photos by using deep learning models. Our design focuses on effectively scaling to accommodate abrupt spikes in user numbers, and we are utilizing Infrastructure as a Service (IaaS) capabilities from Amazon Web Services (AWS) to maximize performance.

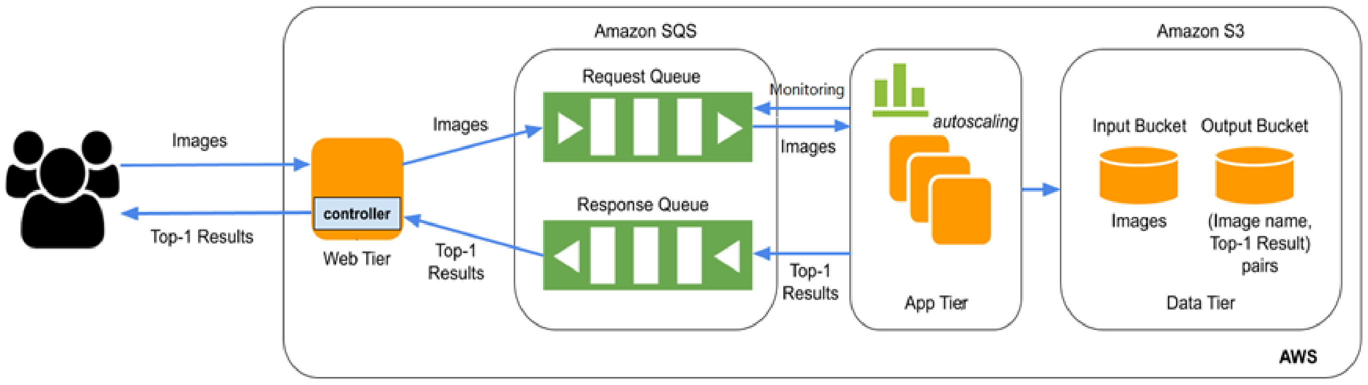
Users can access our application using RESTful services thanks to its architecture. It accepts input photographs and produces results for image recognition by utilizing a number of AWS services for communications, processing, and storage. The creation of an elastic application that can manage several concurrent requests is the main goal of this project. The application should scale out as demand for requests rises, and scale in when demand falls.

To ensure application resilience, cost-efficiency, and user delight, it is essential to create an elastic application that can automatically respond to demand. The program can adjust in real-time to shifting conditions and variable traffic levels thanks to automatic scaling. Utilizing resources efficiently, cutting costs, and improving overall application efficiency are all benefits of this strategy. To ensure that customers can use the application quickly and efficiently, it is crucial to provide constant response times, especially during moments of high demand.

In conclusion, elastic scaling is a critical component of our project's strategy for optimizing user experience and application performance.

2. Design and implementation

2.1 Architecture



Web Tier: The Web Tier acts as the user interface, enabling interaction by allowing users to contribute photographs and get outcomes produced by the App Tier. To manage picture uploads and connect image data and filenames with a SQS (Simple Queue Service) queue data and to upload the images to input S3 bucket, we created this component using Spring Boot. After all uploaded photographs have been sent to SQS, we query the response SQS queue indefinitely to find results and provide them to users.

App Tier: Using SQS, we process the picture data that has been provided by users in the App Tier. The byte data is decoded, transformed into picture data, and then the image then labeled using a predetermined model, and the file name and image classification label are uploaded to another S3 result bucket. The findings are simultaneously transmitted to a response queue where users can view them.

Autoscaling: We developed a scaling-out strategy based on two key parameters: the current number of running instances and an estimate of visible SQS (Simple Queue Service) messages. Our primary goal is to never exceed 20 instances while keeping the instance count as close to the SQS message count as possible. If the number of SQS messages exceeds 20, we prioritize keeping a total of 20 application instances, creating additional instances as needed. This approach reflects a pragmatic, 'greedy' solution aimed at providing services to users as efficiently as possible. Our custom AMI (Amazon Machine Image) streamlines the process even further by allowing the Java application to be executed via a shell script each time an instance boots up, ensuring rapid execution.

Amazon SQS: Amazon Web Services offers the distributed message queuing service known as Amazon Simple queuing Service (Amazon SQS). Even when they are executed in separate places, it facilitates smooth communication between various software components or microservices. SQS queues retain messages until they are processed or removed. We use Amazon SQS in our design to separate the Web Tier and App Tier. Due to its high throughput and best-effort ordering features, we chose the standard queue type.

S3 Buckets: Amazon Web Services Amazon S3 (Simple Storage solution) is a flexible object storage solution. Data of any size, at any time, from any location on the internet, may be stored and retrieved using S3. For our project, two S3 buckets have been created. The first folder, input files, contains picture information and file names that were obtained via SQS. The second, called "output files," saves filenames and labels for picture categorization, making it easier to transfer data to SQS for user access as needed.

2.2 Autoscaling

Scaling in and scaling out are separate processes that take place at the application and web tiers, respectively. In the web tier, incoming requests are routed to the SQS (Simple Queue Service) input. Given the concurrent nature of requests served by different threads in our web application, achieving thread safety was a significant challenge. To address this, we added a parallel main thread in the web tier that runs alongside other Java threads. This thread implements scale-out logic by taking into account variables such as the number of running instances and estimated SQS message counts. Our goal is to limit instance creation to 20 while keeping the instance count in sync with SQS messages. If the number of SQS messages exceeds 20, we prioritize keeping 20 app instances running for efficient user service. This approach is comparable to a "greedy" solution focused on quick service delivery. Furthermore, for efficient task execution, our custom AMI can execute the Java application via a shell script each time an instance boots up.

Scaling in, on the other hand, takes place at the application level. An application instance stores the output of a request in SQS and S3. It continuously checks the SQS for new messages, repeating the process if one is found. When the queue is empty, the instance exits the loop and terminates, which helps with scaling in. One app instance is an exception, as it continuously handles requests irrespective of whether the queue is empty or not.

2.3 Member Tasks

Manogna: Developing up the front-end application, which makes it easier to input images and retrieve results from the App-tier (Machine Learning model), is my main task in this project's position. I came across a number of front-end design options throughout the planning stage. Since I was already familiar with HTML and JavaScript, I first thought of employing them. But after doing a lot of research, I found that Java, more especially the Spring boot library, offers a simple method to communicate with Amazon Web Services components.

After setting up the infrastructure, I went ahead and deployed my code to an EC2 instance. I started by creating a POST method that would receive an input picture folder, process every image inside of it, and send the processed image data to Amazon Simple Queue Service (SQS) and input S3 bucket in order to reproduce the functionality in Java. The program delivers a message to a pre-configured SQS queue when the picture uploads are complete. Important details from this message, such the file name and picture data, are later used by the App-tier.

I developed a method to monitor the SQS response for the results after sending all of the images to it, and I then presented the output. Once everything was set up, I was able to run this application, thoroughly test my Java code, and see the anticipated outcomes.

Avinash: Developing up the app-tier component and effectively integrating it to perform various duties was my major duty. This integration was created to quickly retrieve photos that were first submitted by the web-tier to the SQS queue. The app-tier's duties included transforming the SQS queue's received byte data into useable image data, classifying the resulting photos using a predefined image classification model, and publishing the resulting classification results into an output S3 bucket. The app-tier was also in charge of sending these findings to a response SQS queue. I also took the effort to build an Amazon Machine Image (AMI) with the required app-tier features in order to streamline these processes. This AMI was used as the basis for setting up the launch template that the auto scaling group used. Additionally, I created a system service that allowed the EC2 instances in the auto scaling group to launch or reboot the Spring Boot application (app-tier.jar).

I have tested each of these features to guarantee the system's resilience and dependability. The system was tested using a variety of inputs, and I expanded the tests to include several queries to the image classification model running simultaneously.

Manaswi : I first created an autoscaling group to dynamically scale the application. This procedure began with the development of a launch template in which I specified the AMI to be used. I created the autoscaling group after installing the launch template. To facilitate communication between the web and app tiers, two SQS queues for message exchange were created, along with two S3 buckets for message storage. I have kept a maximum of 20 instances while ensuring that the instance count corresponds to the number of SQS messages. If the number of SQS messages exceeds 20, our priority is to maintain 20 application instances, ensuring efficient service delivery in a "greedy" approach aimed at rapid service provision. Scaling in, on the other hand, is done at the application level. An application instance stores the output of a request in both SQS and S3. It continuously monitors SQS for new messages and repeats the process when one is detected. When the queue is empty, the instance exits the loop and gracefully exits, helping to scale in. One app instance behaves differently, handling requests continuously with a 20-second wait time to ensure efficient polling.

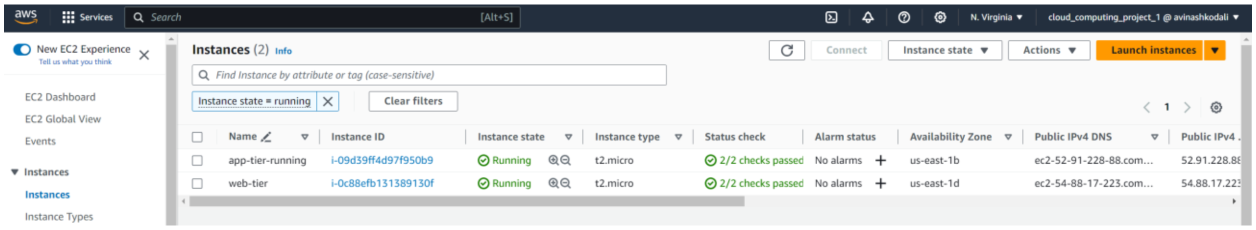
Furthermore, our custom AMI improves task efficiency by running the Java application via a shell script each time an instance boots up.To address the issue of concurrent requests and thread safety, I added a parallel main thread to the web tier, which runs concurrently with other Java threads. This parallel thread was in charge of implementing the scale-out logic.

3. Testing and evaluation

We have used the multi-thread workload generator to send concurrent POST requests to the web tier application running on AWS cloud. During testing we have sent 100 concurrent POST requests to web tier application, we have noticed that the number of messages in the request queue has increased. As the number of request queue messages increased, the number of app tier instances scaled up to process the requests and responses were stored in output S3 bucket and response queue. The number of app tier instances scaled down as the number of messages in request queue to be processed decreased. The web tier application returns the responses from the response queue to the POST request. We have created an UI which shows the results for the image classification.

Results

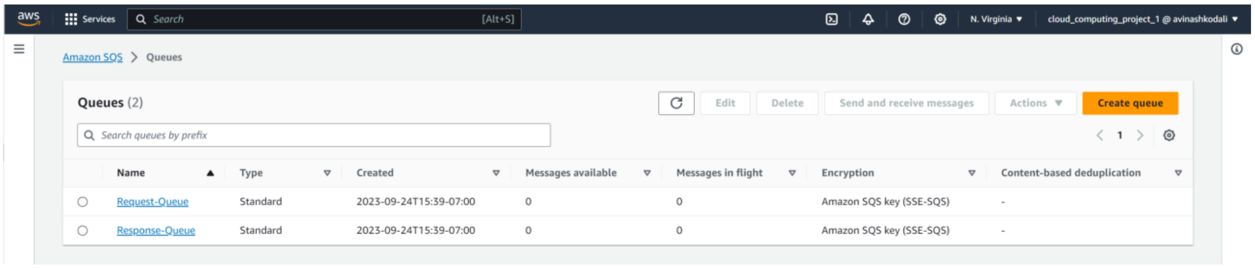
Before Execution



Instances (2) Info

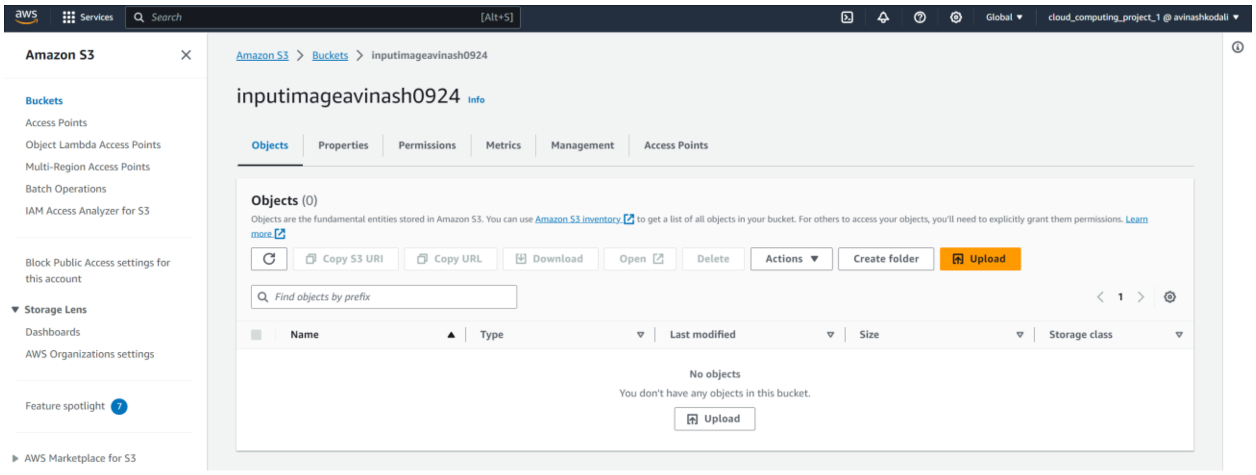
Instance state = running

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4 .
app-tier-running	i-09d39ff4d97f950b9	Running	t2.micro	2/2 checks passed	No alarms	us-east-1b	ec2-52-91-228-88.com...	52.91.228.8
web-tier	i-0c88efb131389130f	Running	t2.micro	2/2 checks passed	No alarms	us-east-1d	ec2-54-88-17-223.com...	54.88.17.22



Queues (2)

Name	Type	Created	Messages available	Messages in flight	Encryption	Content-based deduplication
Request-Queue	Standard	2023-09-24T15:39-07:00	0	0	Amazon SQS key (SSE-SQS)	-
Response-Queue	Standard	2023-09-24T15:39-07:00	0	0	Amazon SQS key (SSE-SQS)	-



Amazon S3 > Buckets > inputimageavinash0924

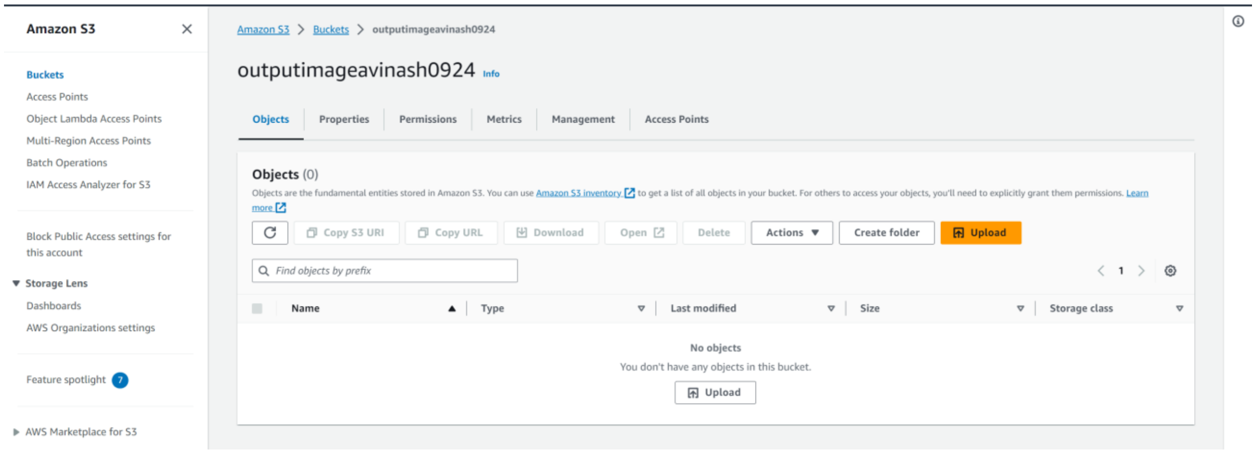
inputimageavinash0924

Objects (0)

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

No objects
You don't have any objects in this bucket.



Amazon S3 > Buckets > outputimageavinash0924

outputimageavinash0924

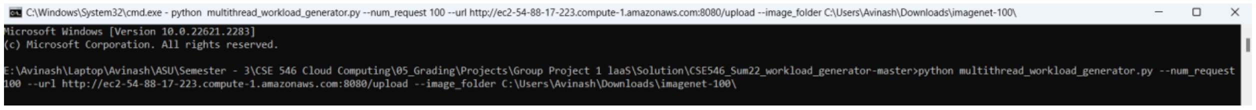
Objects (0)

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

No objects
You don't have any objects in this bucket.

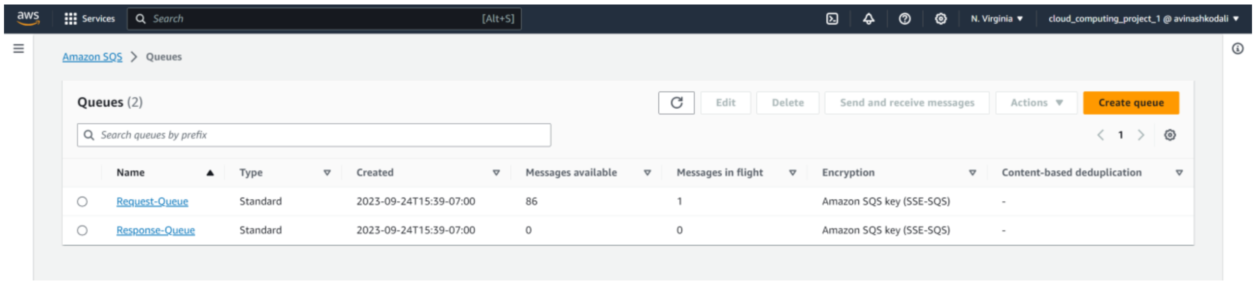
Multi-threaded workload generator



```
C:\Windows\System32\cmd.exe - python  multithread_workload_generator.py --num_request 100 --url http://ec2-54-88-17-223.compute-1.amazonaws.com:8080/upload --image_folder C:\Users\Avinash\Downloads\imagenet-100\
Microsoft Windows [Version 10.0.22621.2283]
(c) Microsoft Corporation. All rights reserved.

E:\Avinash\Laptop\Avinash\ASU\Semester - 3\CSE 546 Cloud Computing\05_Grading\Projects\Group Project 1 laaS\Solution\CSE546_Sum22_workload_generator-master>python multithread_workload_generator.py --num_request
100 --url http://ec2-54-88-17-223.compute-1.amazonaws.com:8080/upload --image_folder C:\Users\Avinash\Downloads\imagenet-100\
```

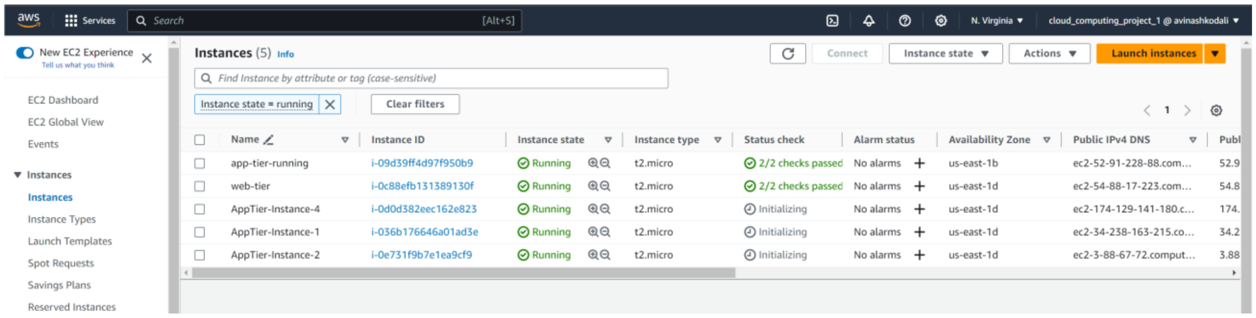
During Execution



Amazon SQS > Queues

Queues (2)

Name	Type	Created	Messages available	Messages in flight	Encryption	Content-based deduplication
Request-Queue	Standard	2023-09-24T15:39-07:00	86	1	Amazon SQS key (SSE-SQS)	-
Response-Queue	Standard	2023-09-24T15:39-07:00	0	0	Amazon SQS key (SSE-SQS)	-

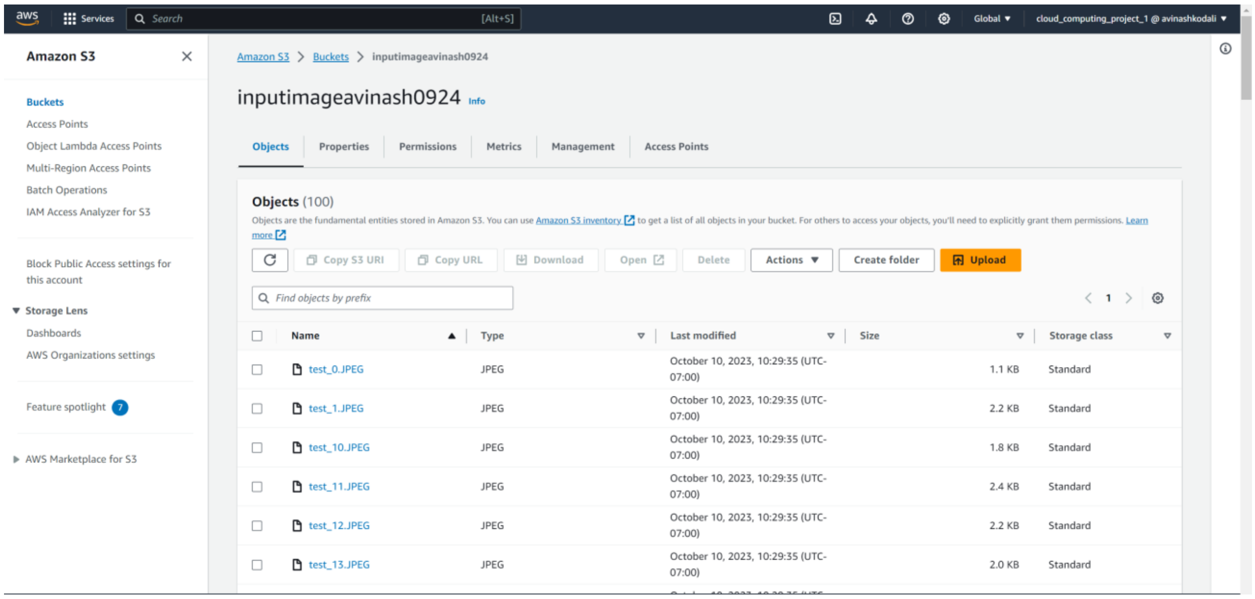


Instances (5)

Instance state = running

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Publ
app-tier-running	i-09d39ff4d97f950b9	Running	t2.micro	2/2 checks passed	No alarms	us-east-1b	ec2-52-91-228-88.com...	52.9
web-tier	i-0c88efb131389130f	Running	t2.micro	2/2 checks passed	No alarms	us-east-1d	ec2-54-88-17-223.com...	54.8
AppTier-Instance-4	i-0d0d382eec162e823	Running	t2.micro	Initializing	No alarms	us-east-1d	ec2-174-129-141-180.c...	174.
AppTier-Instance-1	i-036b176646a01ad3e	Running	t2.micro	Initializing	No alarms	us-east-1d	ec2-34-238-163-215.co...	34.2
AppTier-Instance-2	i-0e731f9b7e1ea9cf9	Running	t2.micro	Initializing	No alarms	us-east-1d	ec2-3-88-67-72.comput...	3.88

After Execution



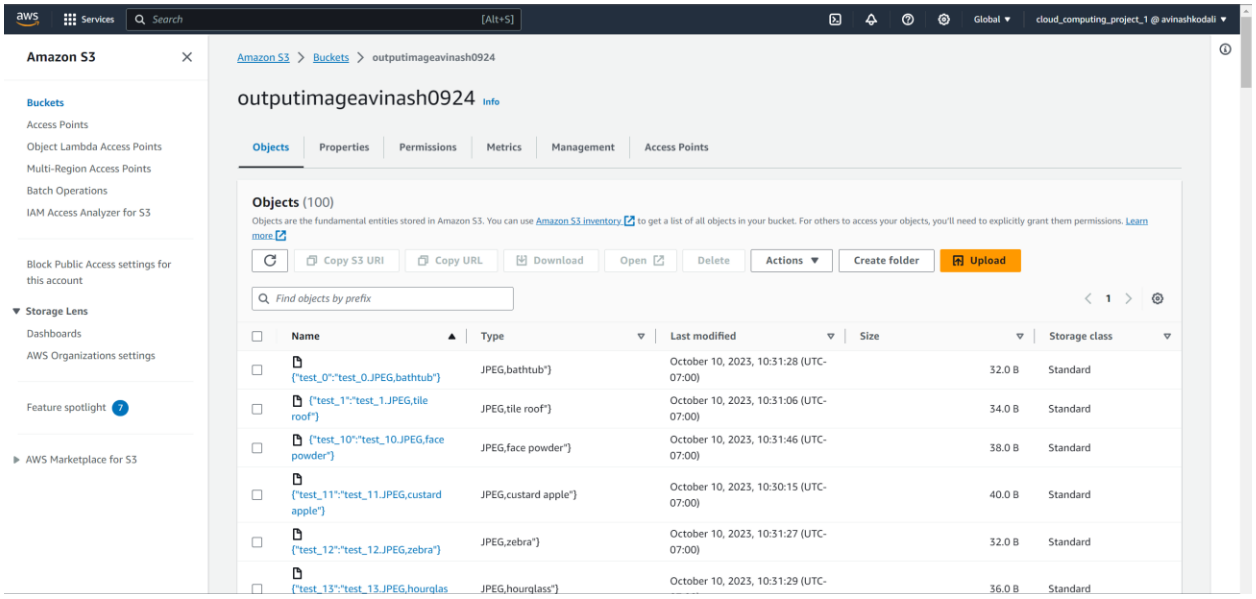
Amazon S3 > Buckets > inputimageavinash0924

inputimageavinash0924

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_0.JPEG	JPEG	October 10, 2023, 10:29:35 (UTC-07:00)	1.1 KB	Standard
test_1.JPEG	JPEG	October 10, 2023, 10:29:35 (UTC-07:00)	2.2 KB	Standard
test_10.JPEG	JPEG	October 10, 2023, 10:29:35 (UTC-07:00)	1.8 KB	Standard
test_11.JPEG	JPEG	October 10, 2023, 10:29:35 (UTC-07:00)	2.4 KB	Standard
test_12.JPEG	JPEG	October 10, 2023, 10:29:35 (UTC-07:00)	2.2 KB	Standard
test_13.JPEG	JPEG	October 10, 2023, 10:29:35 (UTC-07:00)	2.0 KB	Standard



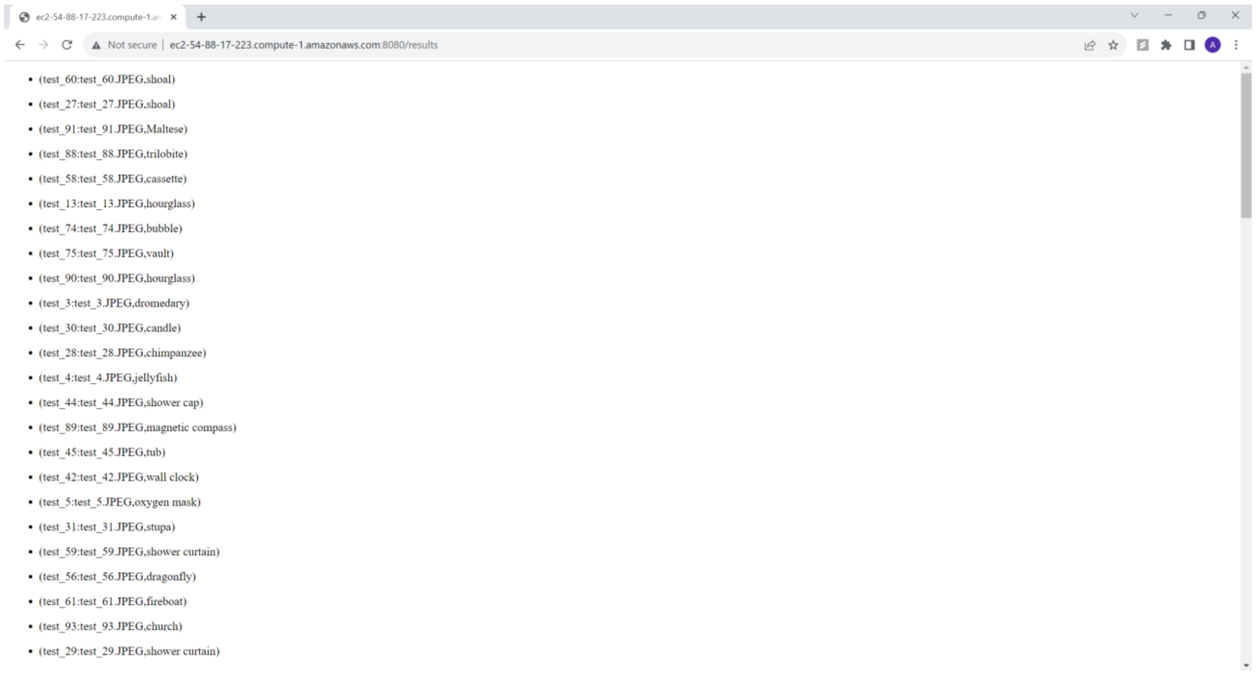
Amazon S3 > Buckets > outputimageavinash0924

outputimageavinash0924

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_0":"test_0.JPEG,bathtub"}	JPEG,bathtub"}	October 10, 2023, 10:31:28 (UTC-07:00)	32.0 B	Standard
{"test_1":"test_1.JPEG,tile roof"}	JPEG,tile roof"}	October 10, 2023, 10:31:06 (UTC-07:00)	34.0 B	Standard
{"test_10":"test_10.JPEG,face powder"}	JPEG,face powder"}	October 10, 2023, 10:31:46 (UTC-07:00)	38.0 B	Standard
{"test_11":"test_11.JPEG,custard apple"}	JPEG,custard apple"}	October 10, 2023, 10:30:15 (UTC-07:00)	40.0 B	Standard
{"test_12":"test_12.JPEG,zebra"}	JPEG,zebra"}	October 10, 2023, 10:31:27 (UTC-07:00)	32.0 B	Standard
{"test_13":"test_13.JPEG,hourglas	JPEG,hourglass"}	October 10, 2023, 10:31:29 (UTC-	36.0 B	Standard


ec2-54-88-17-223.compute-1.a
Not secure | ec2-54-88-17-223.compute-1.amazonaws.com:8080/results

```
• (test_60:test_60.JPEG,shoal)
• (test_27:test_27.JPEG,shoal)
• (test_91:test_91.JPEG,Maltese)
• (test_88:test_88.JPEG,trilobite)
• (test_58:test_58.JPEG,cassette)
• (test_13:test_13.JPEG,hourglass)
• (test_74:test_74.JPEG,bubble)
• (test_75:test_75.JPEG,vault)
• (test_90:test_90.JPEG,hourglass)
• (test_3:test_3.JPEG,dromedary)
• (test_30:test_30.JPEG,candle)
• (test_28:test_28.JPEG,chimpanzee)
• (test_4:test_4.JPEG,jellyfish)
• (test_44:test_44.JPEG,shower cap)
• (test_89:test_89.JPEG,magnetic compass)
• (test_45:test_45.JPEG,tub)
• (test_42:test_42.JPEG,wall clock)
• (test_5:test_5.JPEG,oxygen mask)
• (test_31:test_31.JPEG,stupa)
• (test_59:test_59.JPEG,shower curtain)
• (test_56:test_56.JPEG,dragonfly)
• (test_61:test_61.JPEG,fireboat)
• (test_93:test_93.JPEG,church)
• (test_29:test_29.JPEG,shower curtain)
```

4. Code Explanation

Modules

WebTier

1. REST API is implemented using a spring controller which can handle the POST requests that are generated using a workload generator.
2. Once a post request is received, the web tier sends the image to request SQS and at the same time stores the image to input S3 bucket.
3. Webtier handles the scale in and scale out functionality of AppTier by monitoring the number of messages in the request queue.
4. Webtier fetches the response of the POST request present in the response queue which contains the output of image classification result.

AppTier

1. AppTier fetches the image data from the request queue and processes it using image_classification.py.
2. The output of image classification is stored in the output S3 bucket and sends the result to the response queue.
3. When the request queue has no more messages, the app tier terminates the instance on which it is running resulting in scale in functionality.

AppTier Running

1. It is similar to app tier but the only difference is that this application always runs irrespective of whether the request queue is empty or not.

Execution Steps

1. Navigate to EC2 service using AWS management console, two instances web-tier and app-tier-running are present here.
2. SSH to web-tier and execute nohup java -jar WebTier.jar &, this launches the spring boot application which is capable of handling POST requests.
3. SSH to app-tier-running and execute nohup java -jar /home/ubuntu/app-tier/AppTier.jar &, this launches the spring boot application capable of executing image classification on top of messages received from the request queue.
4. Execute the workload generator which is used to send post requests to web tier applications.
   cmd –
   python multithread_workload_generator.py --num_request 100 --url http://ec2-54-88-17-223.compute-1.amazonaws.com:8080/upload --image_folder IMAGE_PATH
5. The output UI which contains the results of image classification can be accessed using the URL http://ec2-54-88-17-223.compute-1.amazonaws.com:8080/results

References

[1] Amazon Web Services - https://aws.amazon.com/.
[2] Java SDK documentation - https://aws.amazon.com/sdk-for-java/
[3] Auto Scaling - https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-using-sqs-queue.html
[4] Spring Boot - https://spring.io/quickstart