PART-A REPORT

The IEEE paper “A Two-Layer Approach for Reducing Cold Start Delays in Serverless Computing” talks about Serverless Computing, also known as Function as a Service (FaaS) the latest cloud computing model to reduce operational costs and the complexity of system development. Due to this, developers can just breakdown the entire application into small pieces of code called functions containing core logic to do specific tasks thus increasing the level of abstraction from the infrastructure. Despite the benefits of serverless computing, it faces major critical performance challenge, the cold start delay; considered as a delay in preparation of the function execution environment. To tackle this issue, the authors have proposed two main intelligent approaches to determine the best policy for keeping the containers warm based on invocation patterns while optimizing memory consumption. The two-layer approach presented herein leverages reinforcement learning and LSTM prediction to achieve these objectives.

When a serverless function is invoked, it goes through several steps. First, a new container is initialized, which involves setting up the container environment. Then, memory and computing resources are allocated to the container. The function and its dependencies are loaded into the container. The function is then executed, and the results are returned to the user while the logs are saved in the database. These initial steps of initializing the container, allocating resources, and loading dependencies contribute to a delay in response time known as a cold start. After the execution, the container is released, resulting in a scale-to-zero state. Subsequent invocations require repeating the initialization steps, leading to cold start delays.

In the proposed two-layer approach based on Openwhisk platform model, the primary objective of the first layer is to effectively utilize a reinforcement machine learning algorithm to identify the optimal timing for the idle-container window; plays a crucial role in reducing cold start delays and optimizing resource consumption. However, there exists a unique relationship between the number of cold start occurrences and resource usage. Extending the idle-container window results in shorter cold start delays but higher resource consumption. In the second layer, the focus shifts towards minimizing cold start delay time. This is achieved by employing LSTM to forecast future delays and subsequently determining the required number of prewarmed containers based on these predictions.

The efficiency of new approach aimed at reducing cold start delays. The approach was compared to the widely used Openwhisk platform, and various metrics were analyzed to assess its performance. The experiments focused on three key aspects: the time duration that a container remains active after completing a function using platform called idle-container window, the number of cold start occurrences, and the execution of invocations on prewarmed containers. The proposed approach utilized a reinforcement learning algorithm to determine the optimal idle-container window based on invocation patterns, achieving a balance between reducing cold start delays and minimizing resource consumption. The results demonstrated that the proposed approach successfully determined the ideal idle-container window, resulting in fewer cold start occurrences and better control over memory usage compared to the Openwhisk platform. Additionally, the approach outperformed Openwhisk in executing invocations on prewarmed

containers, leading to significant improvements in reducing cold start delays. Overall, the experiments provided strong evidence of the effectiveness of the proposed approach in reducing cold start delays and enhancing the performance of serverless functions, showcasing its adaptability to different invocation patterns and improved resource utilization for a better user experience.

According to authors, to reduce cold start delay among popular platforms is by keeping the containers warm for a fixed period of time after completing a function. This is called the idle-container window. It has 22.65% improvement for execution invocations on prewarmed containers. According to proposed model, in the first layer, by discovering the invocation pattern of the functions by using the reinforcement learning algorithm and determine the idle-container window. In the second layer, the required number of prewarmed containers is determined by the next invocation time prediction of LSTM. This minimizes the number of cold start occurrences and memory consumption by 11.11%, 12.73%, and 4.05% improvement in memory consumption time for the average invocations per hour of 5, 10, and 20, respectively [1].

PART-B REPORT

GitLab Link: https://git.cs.dal.ca/pusuluru/csci-5410-summer-23-b00913674.git

1. Created maven project and added AWS java SDK Dependencies to pom.xml

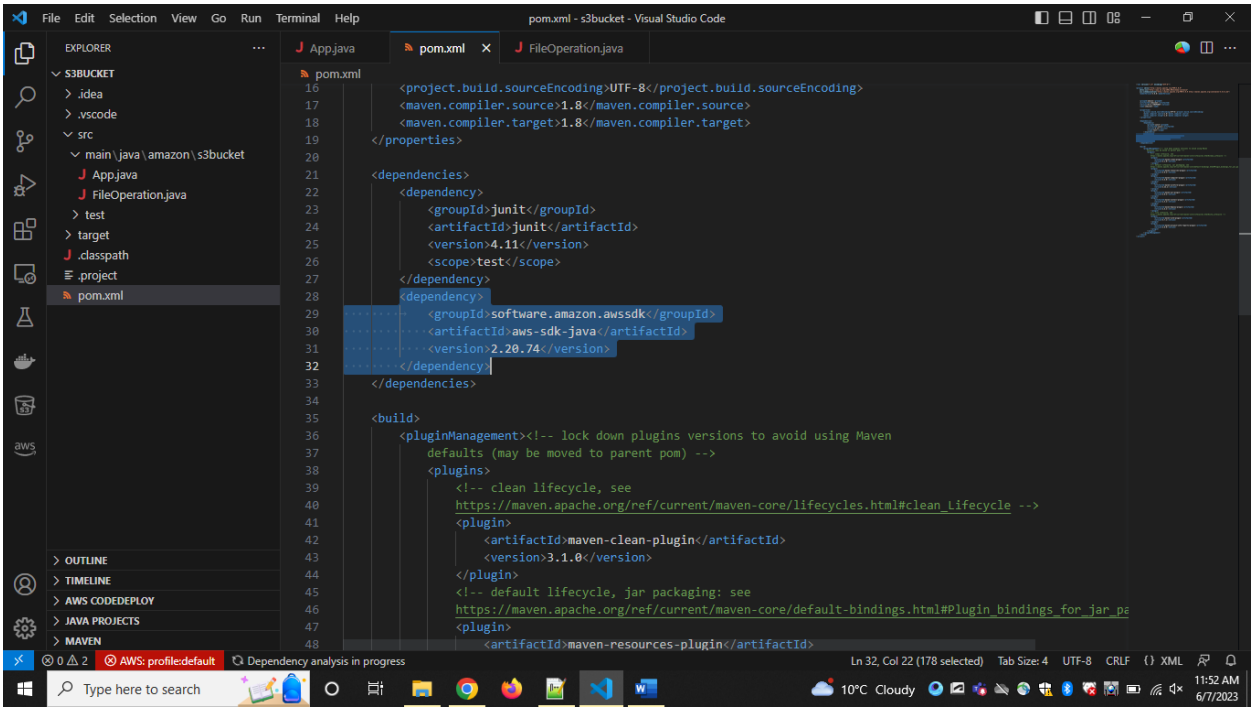

Figure 1: Added AWS SDK Dependency

2. Java Code to add bucket to the Amazon S3 Bucket List by creating S3Client Instance

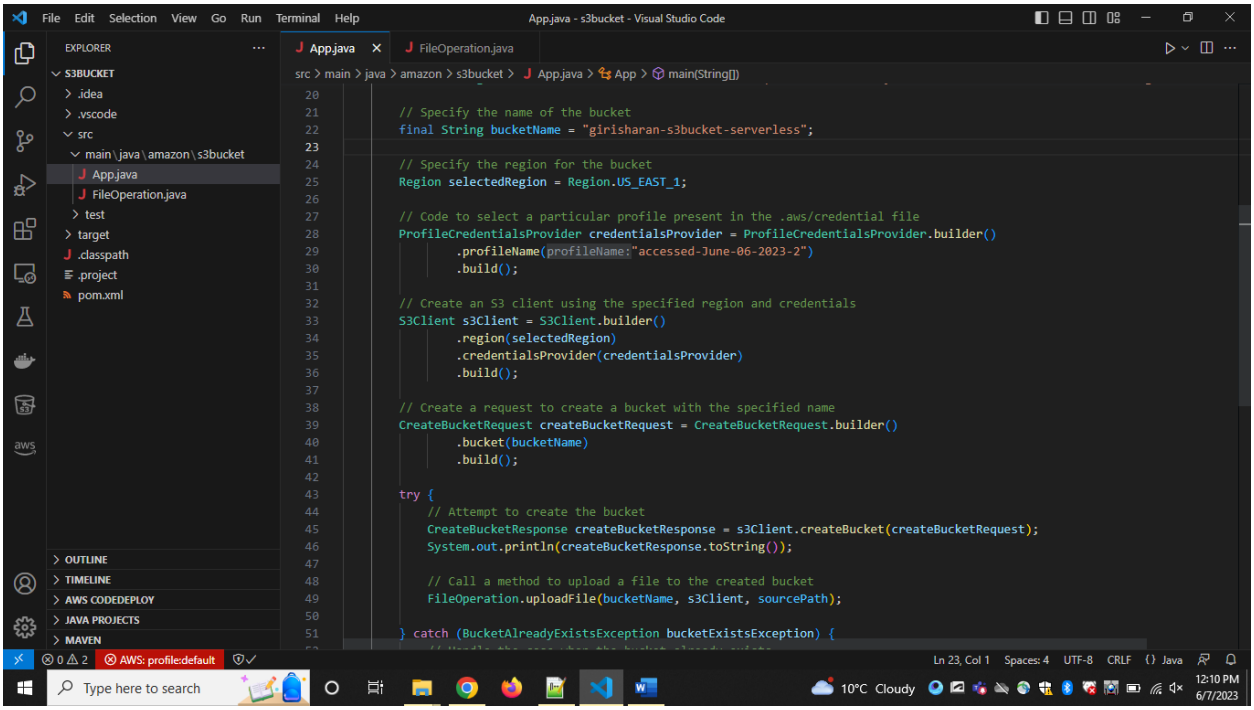

Figure 2: Code to create S3Client instance

3. Bucket successfully created in Amazon Cloud

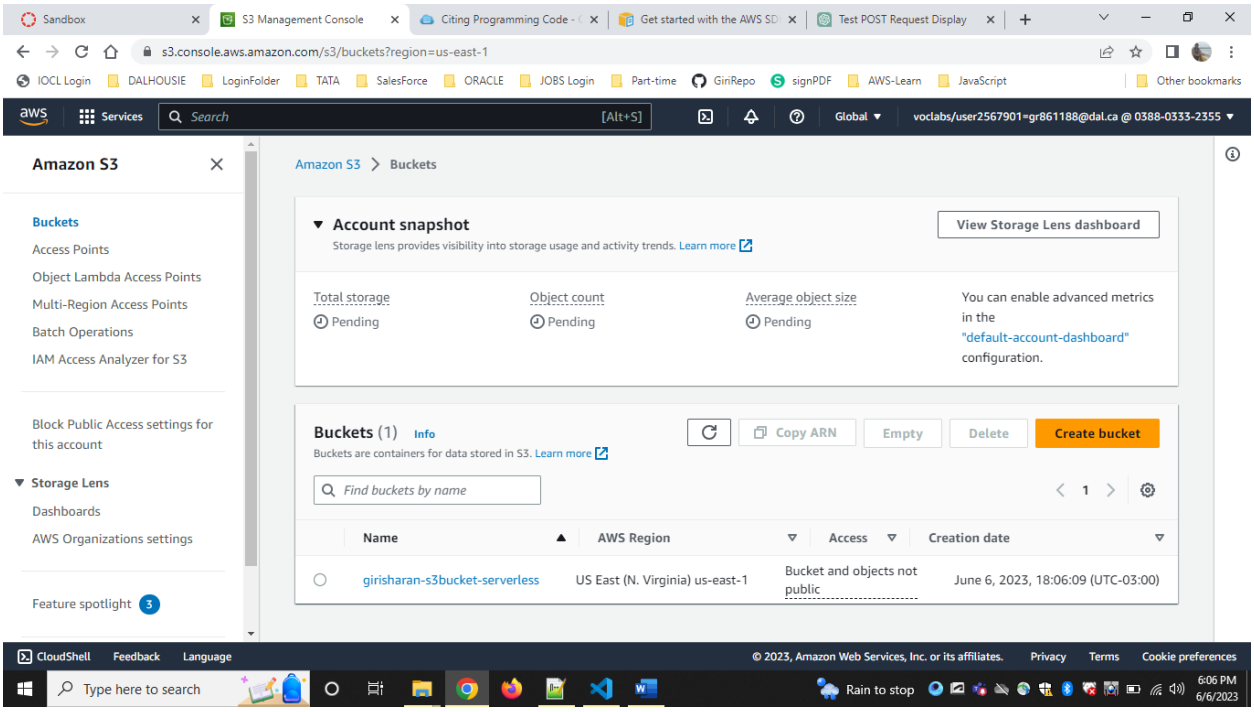

Figure 3: Bucket Created in Amazon S3

4. Initially there was no object in the created bucket

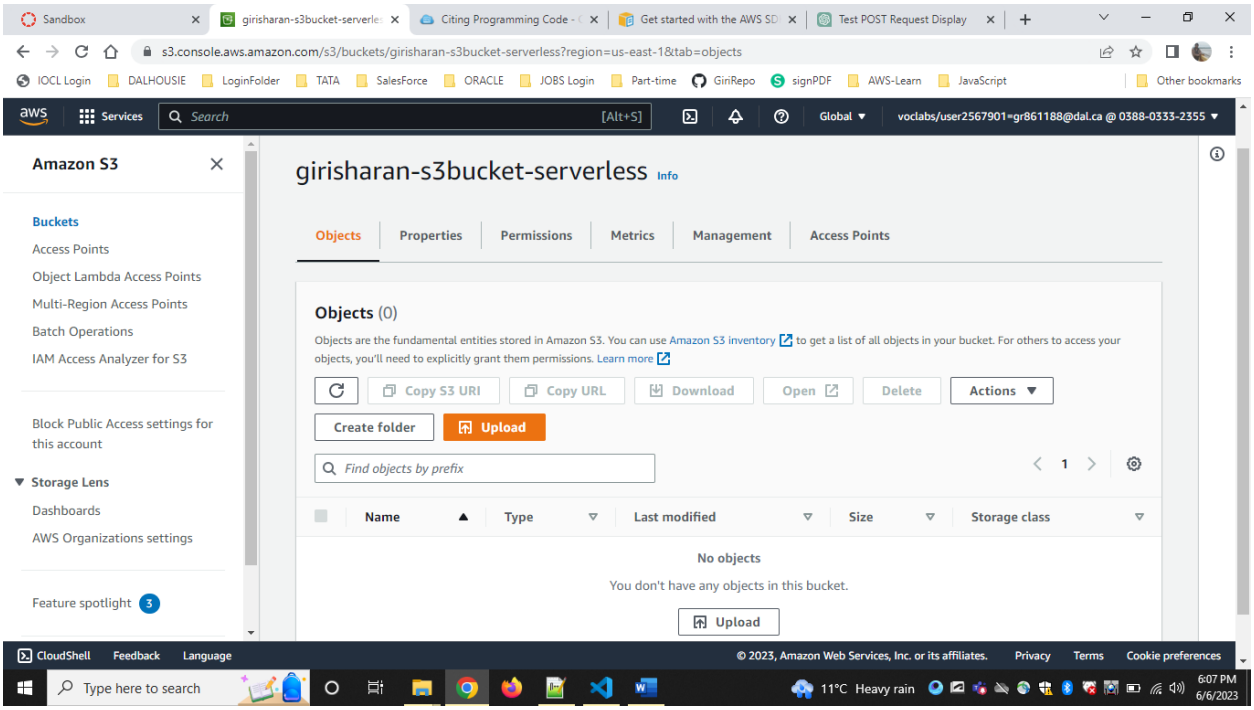

Figure 4: Intially with no bucket in Amazon S3

5. Code to upload file/object (index.html) to the created bucket

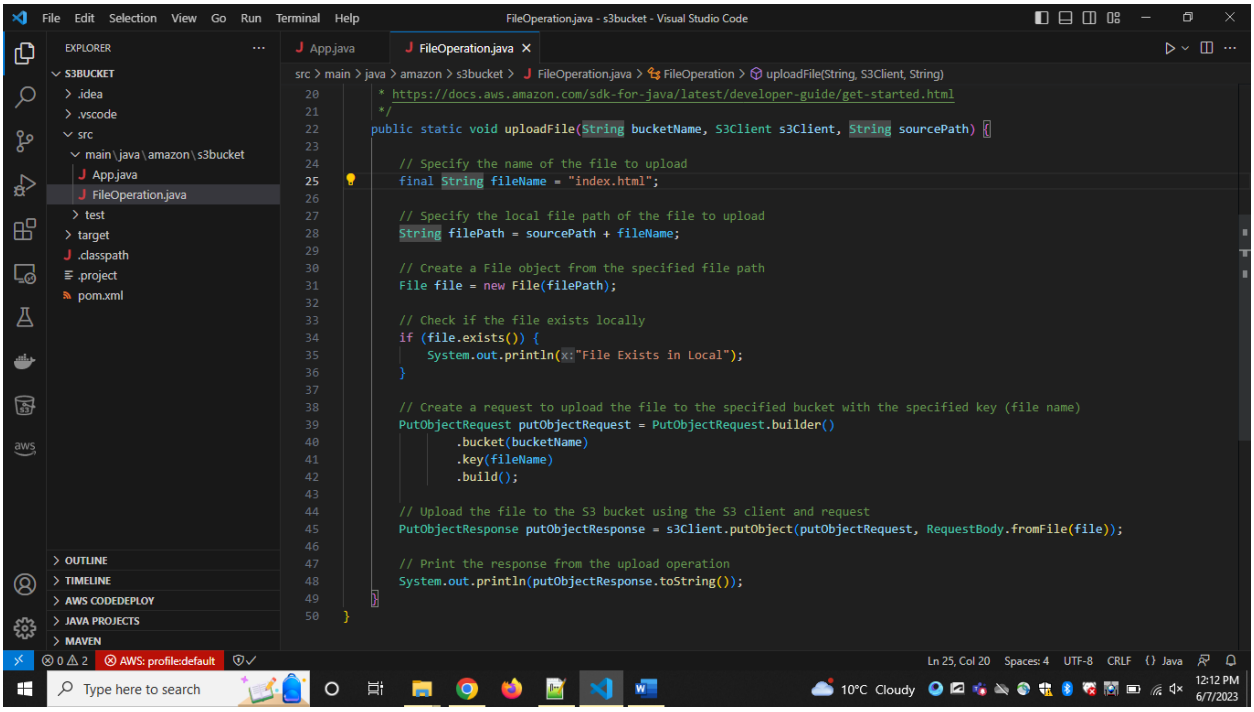

Figure 5: Code to upload object to S3 bucket

6. Object successfully created in the Amazon S3 Bucket

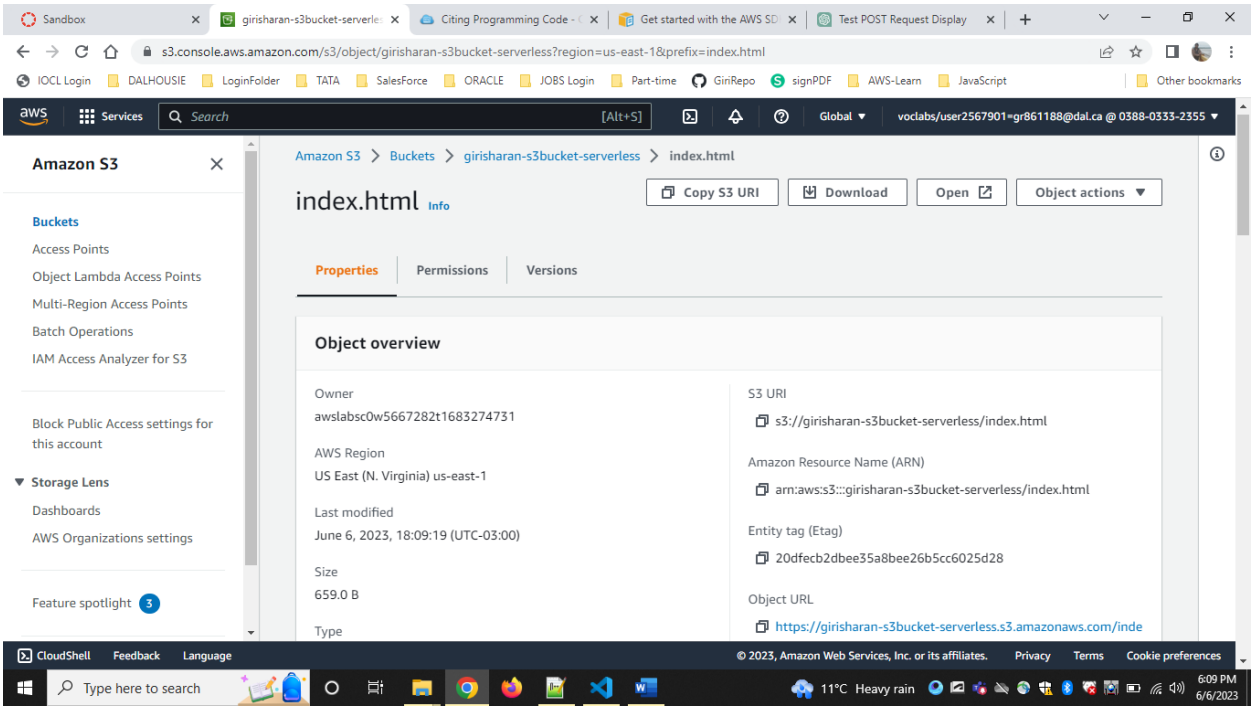

Figure 6: Object in S3 bucket

7. Tried to access object before changing policies and encountered “Access Denied” error

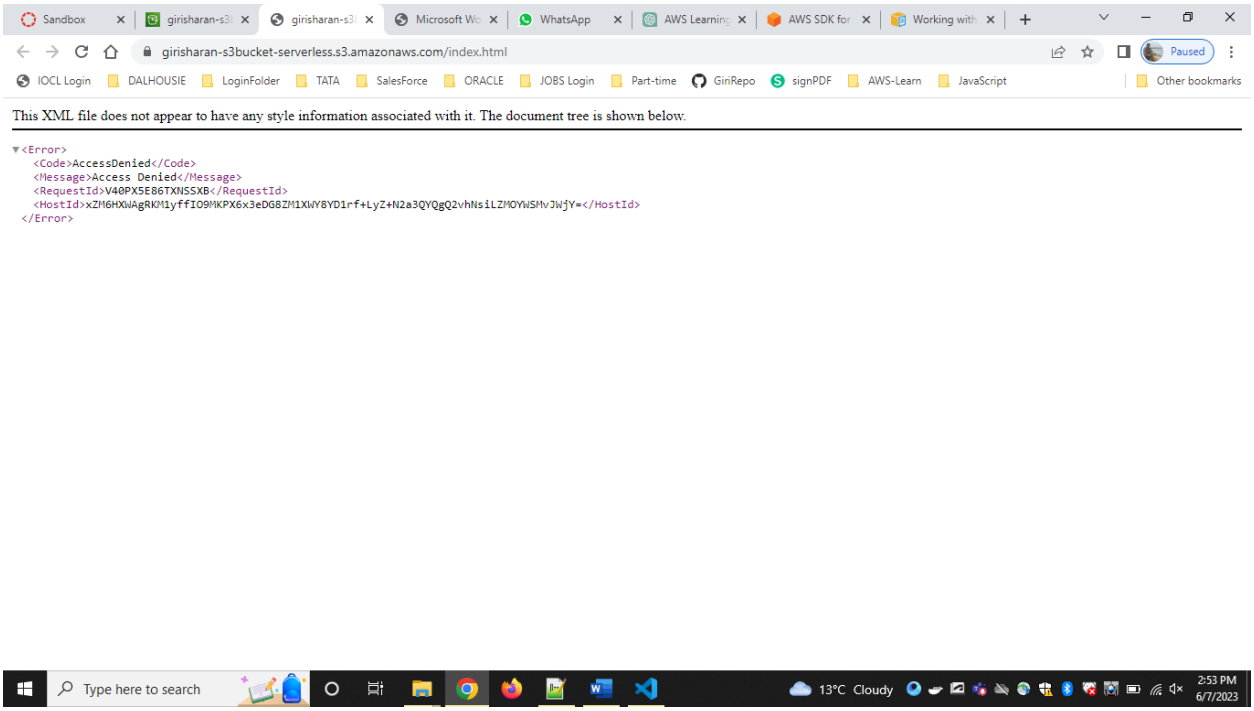

Figure 7: Error when object accessed

8. Now in the Amazon console change bucket permissions such that it is publicly accessible

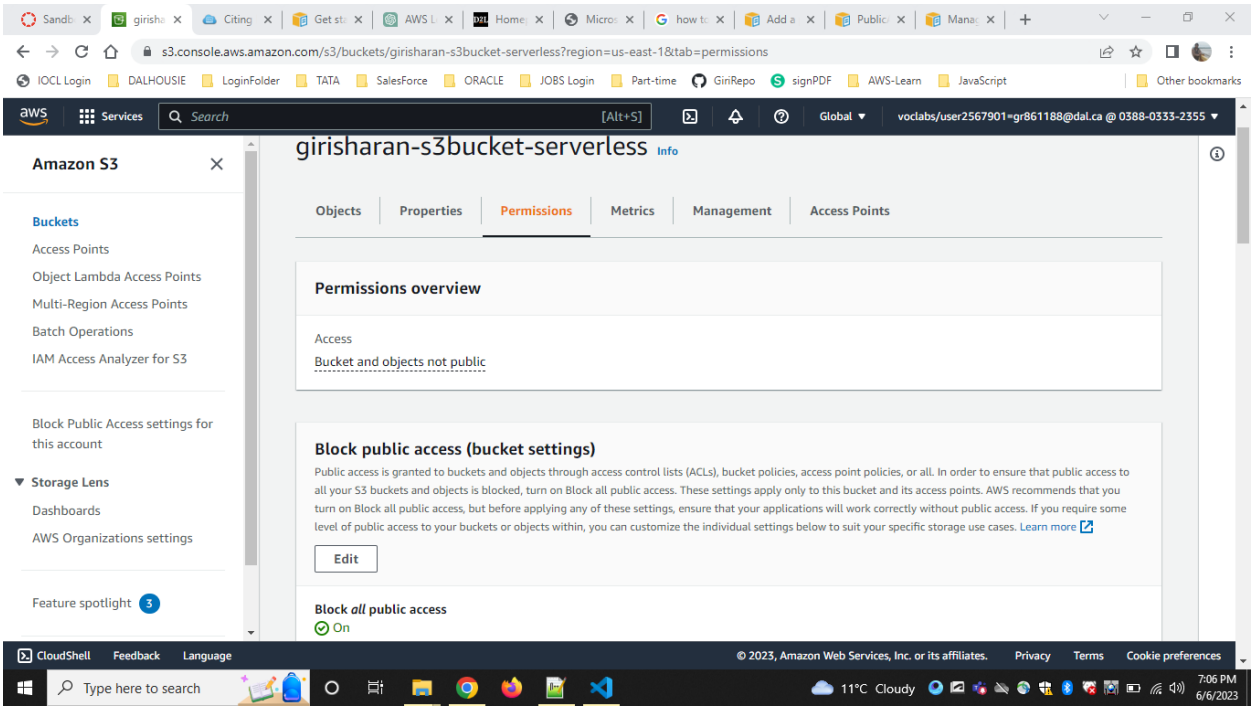

Figure 8: Bucket permissions

9. Change bucket policy to allow public read access to all users

```
{
  "Version":"2012-10-17",
  "Statement":[
    {
      "Sid":"PublicReadGetObject",
      "Effect":"Allow",
      "Principal":"*",
      "Action":["s3:GetObject"],
      "Resource":["arn:aws:s3girisharan-s3bucket-serverless/*"]
    }
  ]
}
```

10. Now go to “properties” and click on “Bucket website endpoint” to host the static website after enabling static website hosting

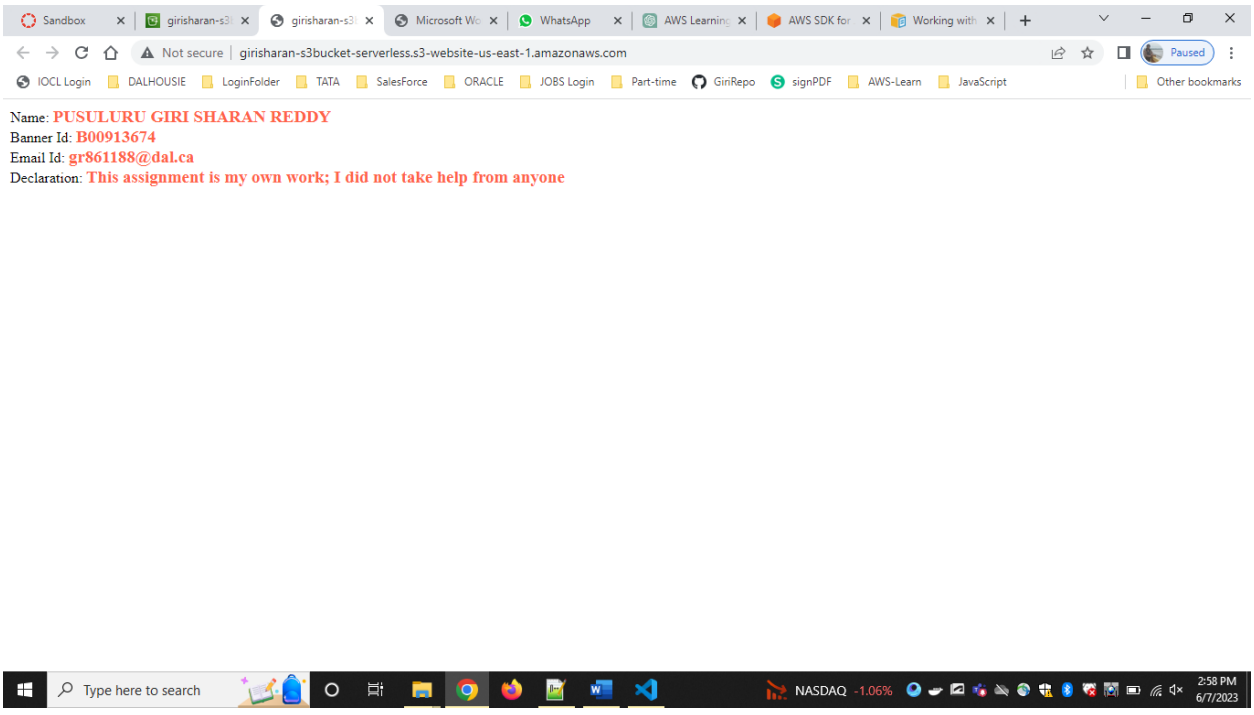

Figure 9: Static Website hosted

Operations performed by me:

Initially I have created a maven project and added the java SDK dependencies required to create the bucket, object in Amazon Cloud Service. Started Lab to get the credentials required to login in to Amazon Cloud. Once the credentials are obtained, copied to .aws/credentials file so that we need to use them in the code directly. Now, with the help of AWS SDK java documentation [1] I have obtained the required knowledge, code snippets to create a bucket and object. Once the code is written without any error, I ran it. There were some errors, exceptions which were resolved, handled respectively. After successful running, bucket and object were created. I tried to host the static website without changing bucket permissions and policies, then I was encountered with "Accessed Denied" error. To solve this, I changed "Block all Policy Access" to off and uploaded a bucket policy for public access under "Permissions". Finally, under "Properties", Static website hosting is disabled by default, change that to enable and mention Index document name and save it. And to see the static website, click on "Bucket website endpoint".

AWS SDK for Java:

The AWS SDK for Java is a collection of libraries and tools that enable application developers to easily interact with various AWS services using Java programming language. Using the SDK, we can build Java applications that work with Amazon S3, Amazon EC2, DynamoDB, and more [2]. To start using the AWS SDK for Java, first we need to include the necessary dependencies in our Java project. Next, we need to configure our AWS credentials this includes providing access key ID and secret access key, which are used for authentication and authorization using Amazon CLI command "aws configure". Next, we need to create an instance of the S3Client class from the AWS SDK for Java. This client provides methods to interact with Amazon S3. Using the S3Client instance, we can invoke the "createBucket" method, specifying the desired bucket name, desired region. This will send a request to Amazon S3 to create a new bucket with the provided details. To upload an object/file to the S3 bucket, we use the "putObject" method of the S3Client. For this method, we need to provide the bucket name, object key (file).

AWS SDK provides additional methods to perform various operations on S3 buckets and objects. This includes deleting objects, deleting buckets, listing objects in a bucket, managing permissions, enabling versioning, setting metadata, and more. By using the AWS SDK for Java, application developers can write efficient and reliable code to create and manage buckets and objects in Amazon S3, making it easier to build scalable and robust applications that leverage the storage capabilities of AWS.

Source Code:

App.java

```
package amazon.s3bucket;

import software.amazon.awssdk.auth.credentials.ProfileCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.BucketAlreadyExistsException;
import software.amazon.awssdk.services.s3.model.BucketAlreadyOwnedByYouException;
import software.amazon.awssdk.services.s3.model.CreateBucketRequest;
import software.amazon.awssdk.services.s3.model.CreateBucketResponse;

public class App {

    /**
     * This method is based on the code provided by Amazon Web Services:
     * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
     */
    public static void main(String[] args) {

        final String sourcePath =
"C:\\Users\\AVuser\\Desktop\\GiriSharanReddy\\dalhousie\\ServerlessDataProcessing
\\AmazonWebServices\\";

        // Specify the name of the bucket
        final String bucketName = "girisharan-s3bucket-serverless";

        // Specify the region for the bucket
        Region selectedRegion = Region.US_EAST_1;

        // Code to select a particular profile present in the .aws/credential
file
        ProfileCredentialsProvider credentialsProvider =
ProfileCredentialsProvider.builder()
                .profileName("accessed-June-07-2023")
                .build();

        // Create an S3 client using the specified region and credentials
        S3Client s3Client = S3Client.builder()
                .region(selectedRegion)
                .credentialsProvider(credentialsProvider)
                .build();
```

```
        // Create a request to create a bucket with the specified name
        CreateBucketRequest createBucketRequest = CreateBucketRequest.builder()
                .bucket(bucketName)
                .build();

        try {
            // Attempt to create the bucket
            CreateBucketResponse createBucketResponse =
s3Client.createBucket(createBucketRequest);
            System.out.println(createBucketResponse.toString());

            // Call a method to upload a file to the created bucket
            FileOperation.uploadFile(bucketName, s3Client, sourcePath);

        } catch (BucketAlreadyExistsException bucketExistsException) {
            // Handle the case when the bucket already exists
            System.out.println("Bucket already exists: " + bucketName);
        } catch (BucketAlreadyOwnedByYouException bucketOwnedByYouException) {
            // Handle the case when the bucket is already owned by you
            System.out.println("Bucket already owned by you: " + bucketName);
        }
    }
}
```

FileOperation.java

```
package amazon.s3bucket;

import java.io.File;
import software.amazon.awssdk.core.sync.RequestBody;
import software.amazon.awssdk.services.s3.model.PutObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;

public class FileOperation {

    /**
     * Uploads a file to the specified S3 bucket.
     *
     * @param bucketName the name of the S3 bucket
     * @param s3Client   the S3 client used for the upload
     */
```

```java
    /**
     * This method is based on the code provided by Amazon Web Services:
     * https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html
     */
    public static void uploadFile(String bucketName, S3Client s3Client, String sourcePath) {

        // Specify the name of the file to upload
        final String fileName = "index.html";

        // Specify the local file path of the file to upload
        String filePath = sourcePath + fileName;

        // Create a File object from the specified file path
        File file = new File(filePath);

        // Check if the file exists locally
        if (file.exists()) {
            System.out.println("File Exists in Local");
        }

        // Create a request to upload the file to the specified bucket with the specified key (file name)
        PutObjectRequest putObjectRequest = PutObjectRequest.builder()
                .bucket(bucketName)
                .key(fileName)
                .build();

        // Upload the file to the S3 bucket using the S3 client and request
        PutObjectResponse putObjectResponse = s3Client.putObject(putObjectRequest, RequestBody.fromFile(file));

        // Print the response from the upload operation
        System.out.println(putObjectResponse.toString());
    }
}
```

REFERENCES:

[1] P. Vahidinia, B. Farahani and F. S. Aliee, "Mitigating Cold Start Problem in Serverless Computing: A Reinforcement Learning Approach," in *IEEE Internet Things J.*, vol. 10, no. 5, pp. 3917-3927, March 1, 2023, doi: 10.1109/JIOT.2022.3165127.

[2] “Get started with the AWS SDK for Java 2.x - AWS SDK for Java,” docs.aws.amazon.com. online [Accessed: 07-Jun-2023] https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html.