Cloud Computing Report

Group Members

Aninda Maulik

Poulomi Nandy

Aditya Das

Dharmalingam Dhayananth

Academic Supervisor

Charlotte Laclau

Mines Saint Etienne

Université Jean Monnet

Acknowledgement

We would like to thank Prof. Charlotte Laclau who guided throughout this project in order to complete this project. We also giving our thanks to the team member who worked hard to achieve our milestone.

Aninda Maulik

Poulomi Nandy

Aditya Das

Dharmalingam Dhayanant

Abstract

Cloud computing is a service of on-demand availability of computer system resources and applications, especially data storage (cloud storage) and computing power, without direct active management by the client.

We have demonstrated some of AWS cloud services by integrating them with our program. In this report we explain detail information about the architecture and the functionalities of the applications and how cloud services were adopted with this project. We followed guidelines and good practices that are been addressed in the AWS document and guided by our professor,

Initially, the design and the project architecture has been proposed with details. Then the stack of the project has been discussed. By following this, the implementation is explored with some screen shot of the actual code.

Introduction

Project architecture

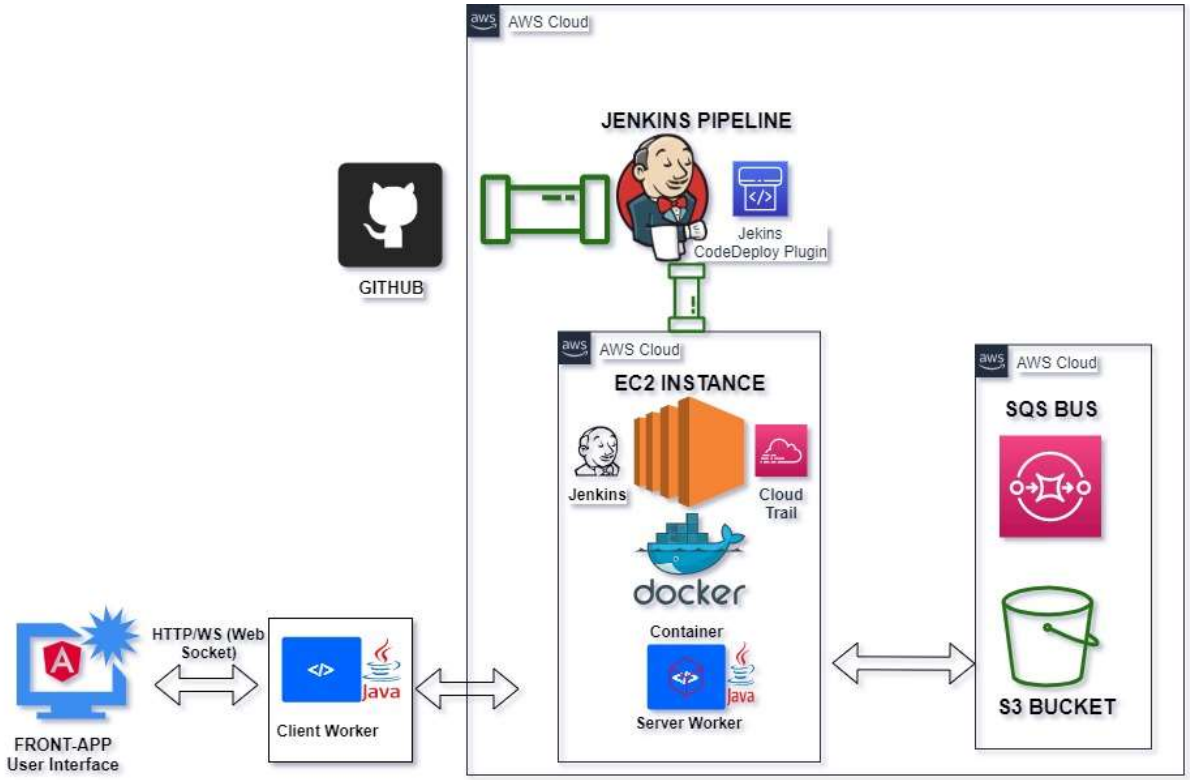

Figure 1: Project architecture

Figure 1, visualise the project architecture of the implementation.

EC2 INSTANCE

An EC2 instance is a virtual machine in Amazon Elastic Compute Cloud (EC2). EC2 is for running application on the Amazon Web Services (AWS) infrastructure. EC2 is a type of Infrastructure as a Service (IaaS). Users can able to select any amount of resources (Storage, Computation power, OS, Number of instances) based on their requirement. (Rouse, Amazon EC2 instance, 2016)

For this project, an EC2 instance has been configured with docker which running on AWS Cloud platform. This Virtual machine act as a main resource of our application. The instance is loaded with Ubuntu OS in order to support the selected stack and technologies. The instance is installed with Jenkins tool, Docker container, Maven with Maven CLI (Command Line Interface) and Java SDK.

Docker

A tool designed to make it easier to create, deploy, and run applications by using containers. Docker is Containers allow a developer to package up an application with all of the parts it needs

Docker container has been installed in the EC2 Instance in order to launch and execute the program. The developed Java applications will be containerized using "Docker Compose" file which exist in the application folder (Further explained in implementation section of this report).

The containers build with OpenJDK Version-11 and the executable version of the application (jar file). The "Server-Worker" application has been containerized and hosted to the docker hub.

Client-Worker

Client Worker is a java application that responsible for handle client requests and responses that are been send from user interface (Front-ap). The Client worker appliocation also runs in the client side. This Java application responsible for two main tasks. First one is to take the request from client and pass it to the relevant SQS REQUEST messaging bus. Second task is to take the message from relevant SQS RESPONSE messaging bus and send the response to the client (front-app user interface). This application does not perform any computation. The main job of this application is to act as a routing program.

There are two categories of request and response.

1. Numeric request: This request contains list of numbers. This numbers will be directly uploaded to the "request SQS messaging bus" with payload. Also, the response for this request will be retrieved from "response SQS messaging bus". The payload contains the results and it will be sent to the front-app.

2. Image request: This request contains an image. This image will be uploaded to the Amazon S3 bucket (to the original folder). Then the "key" of this file will be sent to the "request SQS messaging bus". As a response, the "key" of edited file will be retrieved from "response SQS messaging bus". Then, the edited image will be downloaded from S3 bucket using the "key" and send back to the front-app.

Server-Worker

Server worker also a Java application that is responsible for computation. There are two main tasks of this application. First task is to compute the mean, min, max, median and average from a list of integers. Second task is image processing (Further explained in implementation section). Firstly, the request for either numeric computation or image processing will be retrieved from request SQS queue. The request will be processed, and the response pushed to the "response SQS queue".

The image processing task is involved with Amazon S3 bucket. When there is a request for image processing, the request only contains the key for the image file that has been uploaded to the S3 bucket. The server worker will pull the image from Amazon S3 bucket using the received key. The image will be processed and again uploaded to the Amazon S3 bucket (to the edited folder). Then the key of this edited image file is being pushed to the "response SQS queue".

Jenkins Worker

Jenkins is an open source tool that offers a service for continuous integration or continuous delivery environment for any combination of languages and source code repositories using code deployment pipelines. This allows different ways of configurations using extended plugins, that development organization can automate the processes of deployment and integration of new features to the currently delivered platform. (Heller, 2020)

Jenkin code pipeline has been installed and configured in the EC2 instance in order to automate the code deploy task. The Jenkins has been configured to monitor the code repositories from Github. When there is a new push to the github repository, Jenkins code deploy pipeline will pull the code from relevant repositories and execute the "docker build" based on the given docker file.

Then, the Jenkins will prepare and push the containers to the given docker hub repositories. At the same time, Jenkins will run the prepared containers in the host machine (EC2 Instance). Currently Jenkins assigned with two tasks in order to run two docker containers.

Github

Github is a version control system that has been used to manage the application code. Github keep the history of different version of code files and keep track of modifications. Developers can collaborate and contribute to the development and can easily integrate their work in a centralised repository. (BROWN, 2019)

For this project, there are three repositories were designed.

1. Server-Worker : This repository contains the Java code for “Server-worker” application.
2. Client-Worker: This repository contains the Java code for the “Client-worker” application.
3. Front-app: This repository is responsible for front-end angular application code.

Jenkins will pull the code from repository 1 and 2 respectively.

Amazon S3 Bucket

Amazon Simple Storage Service (typically known as S3) is a cloud storage resource available in Amazon Web Service platform. This is similar to file storage in common computers. S3 offering object storage service and these objects can be accessed using an API that offered by AWS SDK.

(Rouse, 2015)

There are three S3 buckets were configured for this particular project.

1. Image-list-bucket : This bucket will be used to store and retrieve original image file and the edited image files.
2. Custom-logging: This bucket is used to store custom logging text files. These logging files will be prepared and uploaded to the bucket by client and server worker applications
3. Custom-logging-bin: This bucket used to store binary version of “Java logging list” objects. These objects been prepared in order to simplify the process of “Display logging list” in the front-app. These binary data will be converted to JSON (Java Script Object Notation) objects and send to Angular app, then this will be displayed in a table.
4. Aws-logging: CloudTrail is prepared in the AWS platform to monitor complete actions and events that happening in the cloud resources. AWS Cloud-trail will monitor these actions and stores a log file (json format) “aws-logging” bucket.

AWS SQS

Q1. What is Amazon SQS ? What type of queues can you create ? State their differences.

Aws Simple Queue Service (Known as SQS) is a fully managed messaging queue service. The main purpose of this messaging queue is to support decoupled architecture that is main concept of “Micro-Service Oriented Architecture”, “Distributed-System”, and “Serverless Application”. The SQS messages can be sent in any volume and stored for a selected period.

These messages can be modified or deleted directly in the SQS management console. The messages are asynchronous. The software components can send and retrieve messages at any time without any other service to be available. (Chad Schmutzer, Randall Hunt, 2018)

There are two types of message queue available in AWS.

1. Standard Queue: This queue offers three main features such as, maximum throughput, best effort ordering and at least once delivery.
2. FIFO Queue: This queue service designed to deliver the messages in correct sequence in the exact order that they received and guaranteed that messages are processed exactly once.

Different between Standard Queue and FIFO Queue

Table 1: Different between Simple Queue and FIFO Queue.

Standard queue provides best-effort ordering, generally delivered in the same order.	FIFO queues offer first-in-first-out delivery and exactly-once processing: the order is strictly preserved and guaranteed.
Offers high-throughput, due to this reason, it is possible to retrieve duplicated messages more than one time.	Strictly delivered only once, and remains in the queue if user does not delete it.
Standard queue allows unlimited number of transactions per second	FIFO queues are limited to 300 transaction per second.
Standard queue available in all the regions	FIFO queue service is available in limited regions.
Supported by all AWS Services	Not compatible with all the services of AWS (Ex : CloudWatch Events, S3 Event Notifications, SNS Topic Subscriptions, Auto Scaling Lifecycle Hooks).

Front-App

Front-app is a client-side graphical user interface integrated application that has been developed using Angular framework. Angular is a JavaScript-based library which allows developing front side (SPA) Single Page Application. This framework fully supports TypeScript. Also, we use Bootstrap for design our interface with good looking components. Finally, we use webpack to bundle our JavaScript application. This tool will do the job of "uglifying" of "minifying" process and build our front-app.

This application will start the request process. There are two types of request will be sent to the "Client-worker" application

1. List of numbers: This request will consist of list of numbers and expect a response with min, max, median, mean and average.
2. An image file: This request will send a image to the Client-Worker and expect a response with RGB Colour modified image.

HTTP, WS (WebSocket)

The http standard protocol is used to communicate the "client-worker" server application from "front-app". In addition to http, the WebSocket API also used for extended use cases. This WS protocol helps to enable an open two-way interactive communication session instead of stateless communication (HTTP are stateless communication). Using this communication protocol, it is possible to send response from server to client at any time asynchronously. This helps to send response when there is a message in "Response SQS Queue". (MDN contributors, 2020)

Design Diagrams

Sequence Diagram

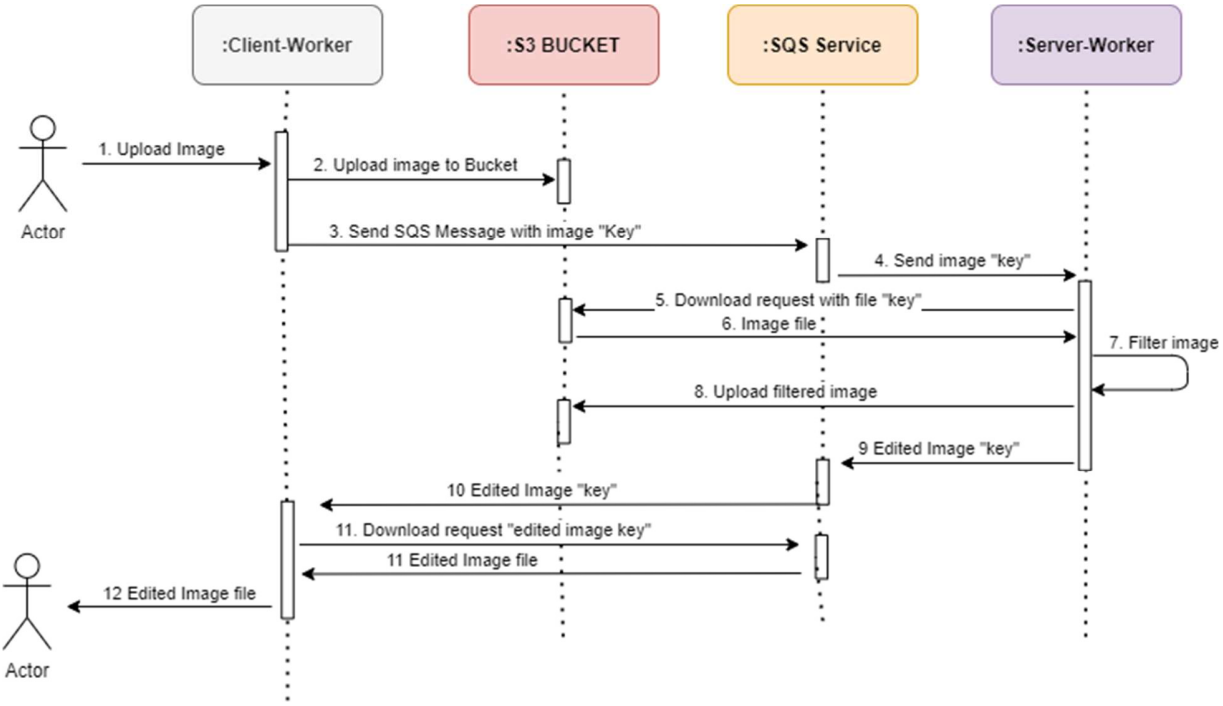

Figure 2: Image processing sequence diagram

Above image shows the functional call and execution flow of image processing event. Initially user uploads a image to client worker using front-app. Client worker will upload the image to the S3 bucket and send a SQS message with the uploaded image “key”. As soon as SQS receives a message, the server worker will get the message asynchronously, since it already subscribed to the SQS. Then the server worker will send a API call to S3 bucket using “AWS SDK” in order to request the specific file that identified using “key”. Then the image file will be downloaded. The downloaded image being processed by “Server-Worker” itself. After, the edited image is being uploaded to the S3 bucket. After that, the “key” of this edited image will be sent to the “Response SQS”. The “Client worker” get the “key” as soon as it sends and it will send a API call to S3 bucket to download the edited image (Note: Edited image and Original Image are in different folder). Then the “client worker” receives the image file. This image file will be sent to the client.

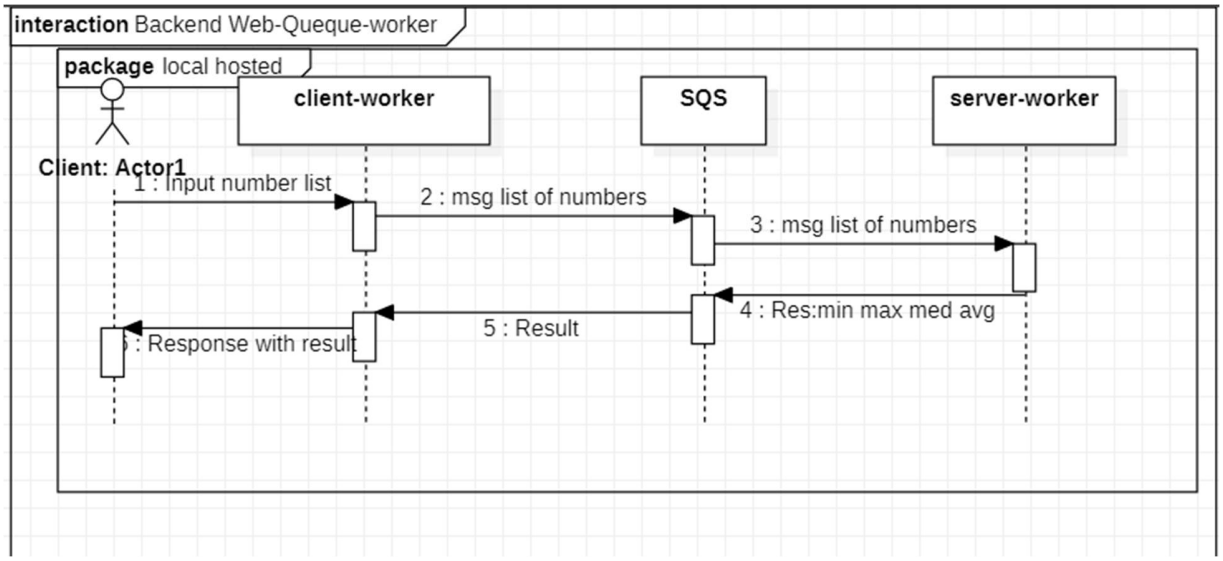

Figure 3: Sequence Diagram for number processing

Above figure shows the shows the functional execution flow of number processing event. Initially user sends a list of numbers to the client-worker from front-app. Then the list of numbers will be sent tot he “request SQS queue”. The “server-worker” will receive the message from “request SQS queue” as soon as when there is a new message event occurred. Then the server worker will process the numbers to get min, max, median and mean. This will be sent as result to the “Response SQS Queue”. This queue is subscribed in the “client-worker” application. So, the application will get the result from SQS and return it to the ‘front-app” using web socket.

Implementation Overview

Configurations of beans

```
@Bean
public AmazonS3Client generateS3Client() {
    AWSSessionCredentials credentials = new BasicSessionCredentials(awsAccessKey, aws
    AmazonS3Client client = new AmazonS3Client(credentials);
    return client;
}
@Bean
public AmazonS3 generateS3() {
    BasicAWSCredentials awsCreds = new BasicAWSCredentials(awsAccessKey, awsSecretKey
    AmazonS3 s3Client = AmazonS3ClientBuilder.standard()
            .withRegion("us-east-1")
            .withCredentials(new AWSStaticCredentialsProvider(awsCreds))
            .build();
    return s3Client;
}
```

Figure 4: S3 Bean creation

```
private String sessionToken ;
@Bean
public QueueMessagingTemplate queueMessagingTemplate() {
    return new QueueMessagingTemplate(amazonSQSAsync());
}
@Primary
@Bean
public AmazonSQSAsync amazonSQSAsync() {
 AWSSessionCredentials credentials = new BasicSessionCredentials(awsAccessKey, awsSecretKey, sessionToken);
  return AmazonSQSAsyncClientBuilder.standard().withRegion(Regions.US_EAST_1)
          .withCredentials(new AWSStaticCredentialsProvider(credentials))
          .build();
}
```

Figure 5: SQS Bean creation

Initially the “S3 Bean” and “SQS Bean” were created using Amazon Java SDK. These beans can be auto injected throughout the program life span. Also, these beans will be created in the application bootstrap (Program initiation period) period. Spring boot dependency injection framework will help to inject these beans to any services or components. In order to create AWS Service beans, it is necessary to pass “aws_access_key”, “aws_secret_key”, and “session_token”. Additionally, regions should be specified to access selected buckets.

Number computation program

Client-Worker

```
@RequestMapping(method=RequestMethod.POST, path= "/num_l" )
public ResponseEntity<String> sendNumberList(@RequestBody final NumberListRequest request) {
    LOGGER.info("Sending the message to the Amazon sqs.");
    if(!request.input.isEmpty()){
        String messageBody=Jackson.toJsonString(request);
        queueMessagingTemplate.convertAndSend(sender_queue_num_list, messageBody);
        sqsService.sendMessage(sender_queue_num_list, messageBody);
        LOGGER.info("Message sent successfully to the Amazon sqs.");
        return new ResponseEntity("Message sent successfully to the Amazon sqs.", HttpStatus.OK);
    }else {
        return new ResponseEntity<>("Number List Empty", HttpStatus.BAD_REQUEST);
    }
}
```

Figure 6: Post methods of number list

Above figures shows how post request of number list is handled in the "client-worker". The application will capture the request and pass the data to SQS message service. This service class will send the number list object to SQS queue. Before sending the object to SQS, it is recommended convert the binary object data to JSON string format. It helps to increase the interoperability. The SQS bean will be used to send the data to messaging queue. Below image shows the implementation of SQS Service class.

```
@Service
public class SqsService {
    @Autowired
    private QueueMessagingTemplate queueMessagingTemplate;

    public void sendMessage(String queueName, String messageBody) {
        queueMessagingTemplate.convertAndSend(queueName, messageBody);
    }

}
```

Figure 7: SQS Service

QueueMessagingTemplate service is provided by AWS SDK, that will be injected with appropriate configuration based on SQS Bean configuration. "convertAndSend" method from this sqsService helps to send the message to relevant SQS Queue. This service is implemented in "client-worker".

```
@SqsListener(value = reciever_queue_num_list, deletionPolicy = SqsMessageDeletionPolicy.ON_SUCCESS)
public void getMessageFromSqs(
          String message,
          @Header("MessageId") String messageId,
          @Header("LogicalResourceId") String logicalResourceId,
          @Header("ApproximateReceiveCount") String approximateReceiveCount,
          @Header("ApproximateFirstReceiveTimestamp") String approximateFirstReceiveTimestamp,
          @Header("SentTimestamp") String sentTimestamp,
          @Header("ReceiptHandle") String receiptHandle,
          @Header("Visibility") QueueMessageVisibility visibility,
          @Header("SenderId") String senderId,
          @Header("contentType") String contentType,
          @Header("lookupDestination") String lookupDestination
          ) {
```

Figure 8: SQS Listener

@SqsListner annotation helps to subscribe to the specific queue. When there is a new message to this queue, the message will be automatically retrieved, and the assigned function will be started to execute. So, this above method start to execute when "server-worker" returns the response with result. The SQS message contains many header information (message id. Sender id etc) that will be used to create log file programmatically. As soon a new message has been retrieved successfully, the message will be deleted from the queue.

```
    }
    ObjectMapper mapper = new ObjectMapper();
    NumberListRequest response=new NumberListRequest();
    try {
        response= mapper.readValue(message, NumberListRequest.class);
    } catch (JsonMappingException e) {
        e.printStackTrace();
    } catch (JsonProcessingException e) {
        e.printStackTrace();
    }
    ResponseMessageModel msg=new ResponseMessageModel();
    msg.message=Jackson.toJsonString(response);
    msg.page_id=2;
    msg.func_id=200;
    msg.message_type="message";
    messagePip=Message_Handler_Singleton.getInstance();
    messagePip.sendMsh(msg);
    LOGGER.info("Successfully Dispatched");
    try {
        logginService.getLogginList();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Figure 9: Response to the front app (number list)

Above figure shows the code of web socket response to the client. The response contains the result of number list, that are min, max, median and mode.

Server-Worker

```
@SqsListener(value = sender_queue_num_list, deletionPolicy = SqsMessageDeletionPolicy.ON_SUCCESS)
public void getMessageFromSqs( String message,
          @Header("MessageId") String messageId,
          @Header("LogicalResourceId") String logicalResourceId,
          @Header("ApproximateReceiveCount") String approximateReceiveCount,
          @Header("ApproximateFirstReceiveTimestamp") String approximateFirstReceiveTimestamp,
          @Header("SentTimestamp") String sentTimestamp,
          @Header("ReceiptHandle") String receiptHandle,
          @Header("Visibility") QueueMessageVisibility visibility,
          @Header("SenderId") String senderId,
          @Header("contentType") String contentType,
          @Header("lookupDestination") String lookupDestination
  ) {
```

Figure 10: Server-worker queue request

The server worker subscribes to the “request SQS queue”, it will receive the request for number list request. All the message headers (Meta data) will be retrieved when there is a new message. As soon a new message has been retrieved successfully, the message will be deleted from the queue.

```
public String getResults(NumberListRequest request) {
    double mean=calculateMean(request);
    double median=calculateMedian(request.input);
    double min=calculateMin(request.input);
    double max=calculateMax(request.input);
    String result="Min {"+min+"}, max{"+max+"}, median{"+median+"}, mean{"+mean+"}";
    return result;
}
```

Figure 11: Calculation method

The get result function will calculate the min, max, median and mode respectively from the given number list. This request will be then sending the “response SQS queue”, that captured by “client-worker” application. This function implemented in server-worker application. Below image shows how the response sent to “response SQS queue”.

```
        }
        final String average=sqsService.getResults(request);
        request.output=average;
        String messageBody=Jackson.toJsonString(request);
        queueMessagingTemplate.convertAndSend(reciever_queue_num_list, messageBody);
        LOGGER.info("Successfully Dispatched to queue");
    }
```

Figure 12: Sending response to SQS queue

Image processing program

Client-Worker

```
@RequestMapping(method=RequestMethod.POST, path = "/uploadT",consumes = {MediaType.MULTIPART_FORM_DATA_VALUE})
public ResponseEntity<Map<String,String>> uploadFileWorking(@RequestPart(value = "file", required = false) MultipartFile files) throws IOException {
        s3bucketService.uploadFile(files.getOriginalFilename(),files.getBytes());
        Map<String,String> result = new HashMap<>();
        result.put("key",files.getOriginalFilename());
        sendToSQS(files.getOriginalFilename());
        return  ResponseEntity.ok(result);
}
```

Figure 13: Upload image client worker

Client worker will receive the image from "front-app" using above post method mapping. The received message will be given to S3 bucket service in order to upload the image to the S3 bucket.

```
        @Autowired
        AmazonS3Client amazonS3Client;
        @Value("${aws.s3.bucket.image_list.name}")
        String defaultBucketName;
        @Value("${aws.s3.bucket.image_list.original.folder}")
        String originalImgFolder;
        @Value("${aws.s3.bucket.image_list.edited.folder}")
        String editedImgFolder;
        public List<Bucket> getAllBuckets() {
            return amazonS3Client.listBuckets();
        }
        public void uploadFile(File uploadFile) {
            amazonS3Client.putObject(defaultBucketName, uploadFile.getName(), uploadFile);
        }
        public void uploadFile(String name,byte[] content)  {
                InputStream is = new ByteArrayInputStream(content);
                ObjectMetadata metadata = new ObjectMetadata();
                metadata.setContentLength(content.length);
                metadata.setContentType("image/jpg");
                metadata.setCacheControl("public, max-age=31536000");
                amazonS3Client.putObject(defaultBucketName, originalImgFolder+"/"+name, is,metadata);
        }
```

Figure 14: Upload original image to bucket

The upload file method will create an input stream of the file and upload the binary data to the bucket. Additionally, the object meta data is been passed to this upload file method that describes about the uploading binary file. The image will be uploaded to the "image-list-bucket" inside "original" folder. And then the key of this image file will be sent to the SQS queue for further process.

```
@SqsListener(value = reciever_queue_image, deletionPolicy = SqsMessageDeletionPolicy.ON_SUCCESS)
public void getMessageFromSqs( String message,
          @Header("MessageId") String messageId,
          @Header("LogicalResourceId") String logicalResourceId,
          @Header("ApproximateReceiveCount") String approximateReceiveCount,
          @Header("ApproximateFirstReceiveTimestamp") String approximateFirstReceiveTimestamp,
          @Header("SentTimestamp") String sentTimestamp,
          @Header("ReceiptHandle") String receiptHandle,
          @Header("Visibility") QueueMessageVisibility visibility,
          @Header("SenderId") String senderId,
          @Header("contentType") String contentType,
          @Header("lookupDestination") String lookupDestination
  ) {
        LOGGER.info("Received reciever image queue message= {}", message);
        ObjectMapper mapper = new ObjectMapper();
        String key="";
        try {
             key = mapper.readValue(message, String.class);
        } catch (JsonMappingException e) {
            e.printStackTrace();
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
        ResponseMessageModel msg=new ResponseMessageModel();
        msg.message=Jackson.toJsonString(key);
        msg.page_id=2;
        msg.func_id=201;
        msg.message_type="message";
        messagePip=Message_Handler_Singleton.getInstance();
        messagePip.sendMsh(msg);
```

Figure 15: Processed image receiver handler

Server-worker will process the given image and upload back to the S3 bucket. The key of the file will be passed to the “response SQS queue” (reciever_queue_image). The above function start to execute when there is a message in reciever_queue_image. The “key” of the image will be retrieved from the queue message and will be passed to the “front-app”. Then the “front-app” will be send another get request to this “client-worker” in order to download the image. This approach is implemented, because there is a limitation when sending a binary image directly to the client using web socket. So, it is better sent image to the “front-app” using standard HTTP/GET request instead of web socket.

```
downloadFilev(key: string): Observable<any> {
  let p = new HttpParams();
  p = p.append('file', key);
  return this.http.get(this.localUrl3 + "s3/download",
    { params: p, responseType: 'blob' as 'json' });
}
```

Figure 16: Front-app download image code

The front-app will receive the “key” of edited image and send a HTTP/GET request to the “client-worker” again to retrieve the edited image. Above figure shows the get image request in “front-app”.

Server-Worker

```
@CrossOrigin
@SqsListener(value = sender_queue_image, deletionPolicy = SqsMessageDeletionPolicy.ON_SUCCESS)
public void getMessageFromSqs( String message,
          @Header("MessageId") String messageId,
          @Header("LogicalResourceId") String logicalResourceId,
          @Header("ApproximateReceiveCount") String approximateReceiveCount,
          @Header("ApproximateFirstReceiveTimestamp") String approximateFirstReceiveTimestamp,
          @Header("SentTimestamp") String sentTimestamp,
          @Header("ReceiptHandle") String receiptHandle,
          @Header("Visibility") QueueMessageVisibility visibility,
          @Header("SenderId") String senderId,
          @Header("contentType") String contentType,
          @Header("lookupDestination") String lookupDestination
  ) {
    System.out.println("SREVER WORKER RECIEVED THE MESSAGE>>"+message);
        LOGGER.info("Received image queue message= {}", message);
        ObjectMapper mapper = new ObjectMapper();
        String key="";
        try {
             key = mapper.readValue(message, String.class);
        } catch (JsonMappingException e) {
            e.printStackTrace();
        } catch (JsonProcessingException e) {
            e.printStackTrace();
        }
        s3bucketService.downloadOrignalFileFromBucket(key);
        String messageBody=Jackson.toJsonString(key);
        sqsService.sendMessages(reciever_queue_image, messageBody);
        LOGGER.info("Successfully Dispatched to queue");
```

Figure 17: Image receiver handler

Above figure shows the implementation of “getMessageFromSqs()” method in “Server-worker” application. The method will receive the uploaded original image “key” the pass it to the “downloadOriginalFileFromBucket()” method. This method will be downloading the original image from bucket and proceed with image filtering. Then send upload it to the bucket again. At last, the function will return the edited image “key”. This key will be sent to SQS queue. The implementation of “downloadOriginalFileFromBucket()” and image processing method shown in below figures.

```
public byte[] downloadOrignalFileFromBucket(String name) {
    try {
        Thread.sleep(500);
    } catch (InterruptedException e1) {
        e1.printStackTrace();
    }
    boolean found=true;
    while(found) {
        List<String> objs=getObjectslistFromFolder(defaultBucketName, originalImgFolder);
        for (String f:objs) {
            if(f.contains("original/"+name)){
                found =false;
        }}
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    try {
    byte [] originalImage=getFile(name);
    InputStream is = new ByteArrayInputStream(originalImage);
        byte[] result=editImageV2(is);
        uploadEditedImageToBucketV2(result, name);
        return result;
    } catch (FileNotFoundException e) {
        e.printStackTrace();
        return null;
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
    catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

Figure 18:Download_image_from_original_bucket method implementation

As described above, this method will download the original file and pass it to the edit image function. The function will change the pixel colour and return it. Then the edited image will be uploaded to the “edited” folder in the S3 bucket. At last the key of this edited image will return to the controller method. Controller method will send a SQS message wit this “key”. So the client worker will proceed further with this key.

Web Socket Configurations

```
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(new SocketTextHandler(), "/user").setAllowedOrigins("*");
        registry.addHandler(new SocketTextHandler(), "/init_connection").setAllowedOrigins("*");
    }

}
```

Figure 19:Web socket config

Above figure shows the initiation of web socket. This allows "front-app" to create a open two-way connection between client (front-app) and the server.

```
public class Message_Handler_Singleton {
    private static Message_Handler_Singleton obj=null;
    private Object session;
    private void Message_Handler_Singleton(){
    }
    public static Message_Handler_Singleton getInstance(){
        if (obj == null)
            obj = new Message_Handler_Singleton();
        return obj;
    }
    public boolean initiate(Object object){
        try {
            this.session=object;
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }
    public WebSocketSession getSession(){
        return (WebSocketSession) session;
    }
    public void sendMsh(Object message){
        try {
            ObjectMapper objectMapper = new ObjectMapper();
            String jsonString;
```

Figure 20: Web socket singleton session handler

Above figure shows the code of "Singleton" object implementation to handle web socket session. This singleton object is being used throughout the application to send message to the connected client (front-app).

```
public class ResponseMessageModel {
    public String message;
    public int code;
    public String status;
    public Object arg;
    public int page_id;
    public int func_id;
    public String message_type;

}
```

Figure 21: Web socket response model

The above code sample shows the web socket response model. The client app will receive the response with above indicated properties to support its functionalities.

EC2 Instance Configuration

Jenkins Configurations

Jenkins has been installed in the EC2 instance and configured to run on port 8585. So, the Jenkins dashboard can be accessed through browser by entering EC2 DNS name and the port number. Using this dashboard, the following Jenkin work has been assigned.

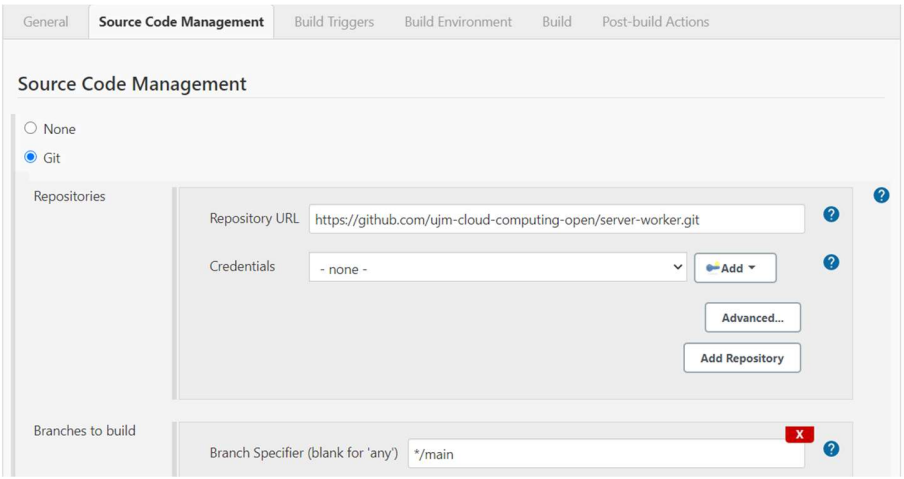

Figure 22: Jenking work: assigned repo

The github repository and the branch has been pointed in the Jenkins where it can fetch the code and start further processing.

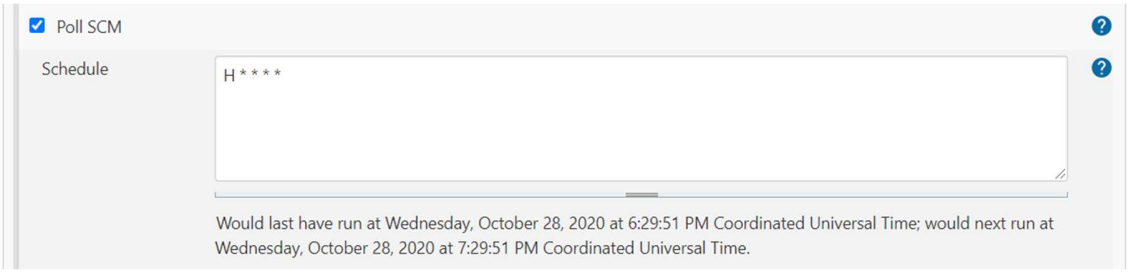

Figure 23: Poll SCM trigger

The build schedule has been configured to check any latest commit to the repository in each 01-hour time interval. If there is any new commit, then the Jenkins will pull the code again and run below build steps again. This helps to automate the complete steps without any manual work.

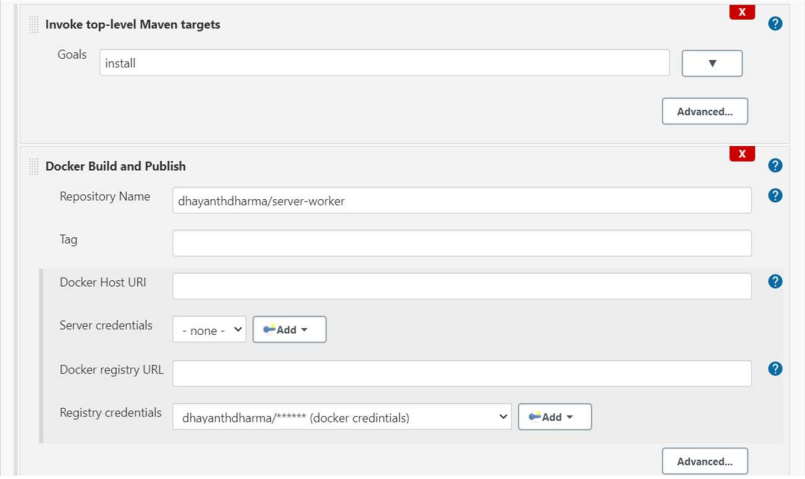

Figure 24: Build steps of Jenkins

The build steps are specified in Jenkins configuration as shown in above figure. Initially "maven install" command will be invoked on top level Java class. This will create the executable .jar file in the target folder.

Second step is to build the docker image. For this purpose, Docker compose file has been added in the root of the application. The contents of this file explained in following pages. This docker file is used to build docker image. Jenkins will build the docker container and push it to the specified repository in "Docker hub" where it can be downloaded anywhere in the world.

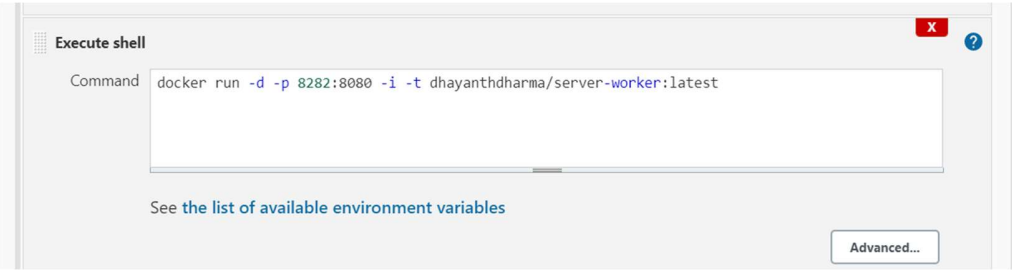

Figure 25: Docker run command for server-worker application

At last the "docker run" command has been specified. This will check the local docker container named "dhayanthdharma/server-worker:latest", and run it. Also, it will bind container port 8080 with host machine port 8282 (Note: "Server-worker" application configured to run on port 8080).

In conclusion, the “Server-worker” application has been deployed in EC2 instance successfully. “Front-app” and “Client-Worker” applications are running locally in user machine. Both applications communicate using SQS. This is known as “Web-Queue-Worker-Architecture”.

Q3: In which situations is a Web-Queue-Worker architecture relevant ?

The web-queue-worker architecture is a cloud-based architecture that is developed by considering decoupled request approach. This is typically implemented using managed compute service on a cloud platform (AWS Cloud, Azure Cloud Service). There are two main components of this architecture, which are client that serves “**client”** requests, and a “**worker**” that performs long running, resource-intensive tasks or batch processing. The “Client” communicates to the “Worker” through a asynchronous message queue.

This architecture is relevant to certain use cases as described below,

- Batch processing
- Applications which processing some long-running computations or operations
- Data mining or Classification using Machine Learning oriented tasks (Which possibly takes longer to get result)
- When there is a need to use managed services, rather than infrastructure as a service (IaaS)
- To manage highest number of requests for long running processes. (Ex: Image processing application could be hosted as a worker. So, when too many client requests for an image processing task, worker will longer respond. So, clients can communicate to worker via a web queue, and clients does not wait for response).

(Microsoft Docs, 2019)

Q2: What is the difference between a resource and a client ?

Client	**Resources**
Gives low level service access	API Based Object oriented implementation. High level access.
Generated from **Service Description**	Generated from **Resource description**
Generally maps with one of the service API.	Contains actions and operations for selected resources.
Exposes botocore client to the developer	Uses Identifiers and attributes of AWS resources.
	Allows to access sub resources and collections of AWS Resources.

(Priyaj, 2019)