CSE 546 — Project 1 Report

Prasidh Aggarwal, Revanth Suresha, Shriya Srinivasan

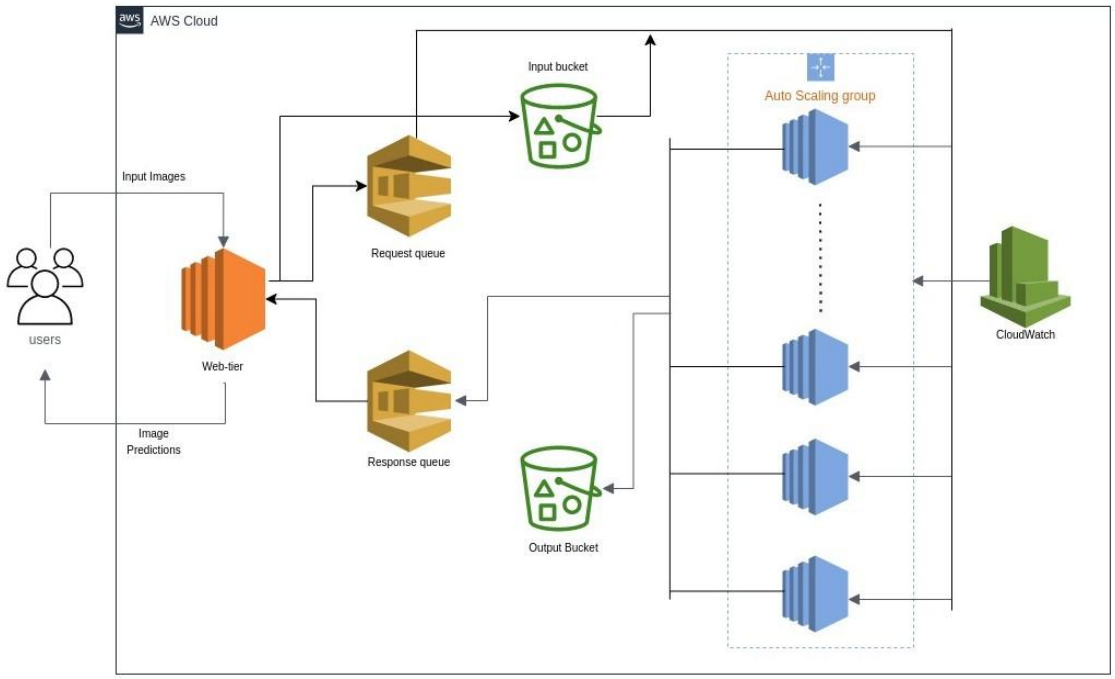
1. Problem statement

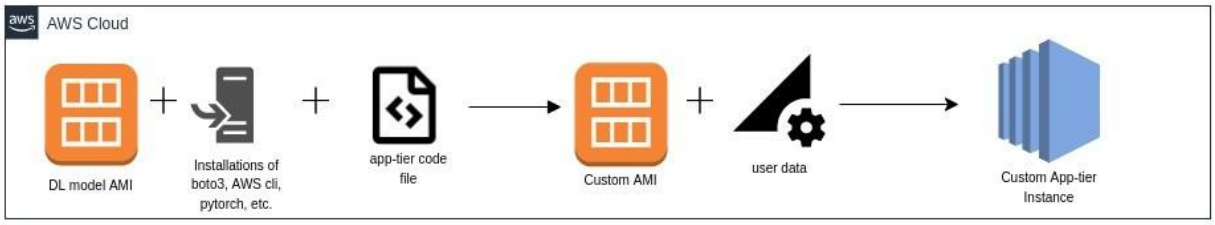
Scalability issues often plague applications deployed on local or isolated servers, necessitating manual scaling and increasing hardware and operational costs. Cloud computing providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform offer load balancing and auto-scaling solutions to address these challenges. In this project, we aim to utilize AWS resources to create an IaaS-based image recognition system that can efficiently handle multiple requests concurrently. We aim to develop an elastic application that can automatically scale up and down based on demand, optimizing operational costs while ensuring the rapid processing of client requests. This project will showcase how cloud computing can lower operating costs and enhance service delivery, offering valuable knowledge and techniques to create future cloud-based applications.

Specifically, an IaaS-based approach was used for this application for the benefits of scalability, flexibility, cost-effectiveness, and accessibility.

2. Design and implementation

2.1 Architecture





The major components of the architecture include:

1. Two S3 buckets, one each for image requests (uploaded by the user) and responses.
2. Two SQS queues, one for request messages and another for response messages containing the file name and the classification results.
3. An Auto Scaling group responsible for scaling the entire application in and out.
4. Two CloudWatch alarms to enable/trigger scaling in and scaling out.
5. Custom AMI to launch EC2 instances made of the one provided (including the ML model); additionally, installations are done for boto3, AWS CLI, etc.
6. Web Tier application, responsible for pushing the images to the request S3 bucket, pushing the messages to the request SQS queue, polling the response SQS queue, and deleting the messages from the response queue.
7. App Tier application, responsible for polling the response SQS queue, downloading images from the request S3 bucket, sending classification results to the response SQS queue, sending response to the response S3 bucket, and finally deleting the messages from the request queue.

2.2 Autoscaling

An autoscaling group is created, connected to two-step scaling policies. Two alarms are designed, one for scaling in and the other for scaling out.

The scale-out approach works as follows: It monitors the size of the request SQS queue and the number of instances in the EC2 Auto Scaling group. It then triggers an alarm when the number of messages in the SQS queue exceeds the number of instances in the Auto Scaling group. The values are linear in the form of steps. If the value breaches 20, then the max number of instances are spawned, i.e., 20. When triggered, the alarm will execute a scale-out action specified in the Alarm Actions property. This action is used to scale up the number of instances in the Auto Scaling group to handle the increased workload.

The scale-in policy works as follows: It monitors the size of the request SQS queue and the number of instances in the EC2 Auto Scaling group using our custom metric. It has set thresholds to trigger the scale-in alarm when the same scale-out metric is in negative values. The values specified are linear, and we also ensure there is one minimum app tier running in the end, even though there are no requests in the SQS queue. For our workload of 100 requests,

the response queue empties fast, and the request queue fills up to 100 very quickly. The instances scale out as soon as the first alarm is triggered and scale in as soon as the response queue size gradually decreases.

2.3 Member Tasks

Prasidh Aggarwal (paggar10) – Chiefly responsible for coding the web tier part of the application. Work included developing a java application that does five main functions: Upload the images to the request S3 bucket, send the messages with file names as the message body to the SQS request queues, continuously poll the response SQS queue for messages with message body as “file-name, classification-result,” delete the messages which have been polled from the SQS response queue, and finally, return the output to the workload generator along the lines of which image was uploaded and what was the image classification result for that image. Also responsible for deploying the web_tier on an EC2 instance, ensuring it has appropriate roles and permissions to do the required operations.

Shriya Srinivasan (ssrin103) – Main responsibilities included coding the app tier part of the ecosystem. Work included developing a python application that does seven main functions: Poll the SQS request queue continuously to look for any new messages, use the request messages to download the respective images from the request S3 bucket, download the images locally, perform image classification on them using the model provided, delete the message from the request SQS queue, send the response message with “file-name, classification-result” in the message body to the response SQS queue, upload the image with the classification result to the response S3 bucket.

Revanth Suresha (rbangal5) – Main responsibilities included setting up all AWS resources as listed. IAM: users, groups, roles, policies, account aliases. S3: Request bucket, Response bucket. SQS: Request queue, Response queue. AMI: Custom AMI with image classification model. Auto Scaling Group, including launch Configuration with user data to automate the installations of different modules on app tiers and run the app.py files. Auto Scaling policies: Scaling in and Scaling out. Cloud Watch alarms: Alarm for Scaling in, Alarm for Scaling out.

Apart from these individual tasks, every team member was involved in designing, implementing, developing, and testing the application.

3. Testing and evaluation

We ran the web and app tiers for the initial testing by providing them with one image request through the workload generator. We were able to receive the correct result but with some formatting issues. After fixing those formatting issues, we tested our application with 20-30-40 and so on requests, still not using the multi-thread generator. After achieving correct results from those requests in under 2-3 minutes, we moved to the multithreaded workload generator, where we started seeing significant issues. We realized that our thread sleeping times needed to be corrected and that it was taking too long for 100 requests to go through. This was when

we assigned one separate thread responsible for polling the SQS response queue rather than all threads polling the queue separately.

Post this phase of testing; we deployed our changes to EC2 instances after doing the necessary java and python module installations. Testing stopped due to many blockers introduced via the user data, such as permission-denied errors. After fixing those issues by modifying the user data scripts appropriately, we tested our application end to end by passing the multithreaded workload generator, our public IP URL of the web tier, and getting 100 responses back in under 2 minutes. The catch, however, was that our auto-scaling group(using target-tracking policies) defined the minimum and desired capacities as 1, due to which one app tier was initially running. This led to one instance of app-tier finishing the entire workload of 100 requests even before the other instances scaled up according to demand.

Even though it was a proper implementation, we changed from target-tracking to simple/step scaling policies. This also meant changing our metric from m1/m2 to m1-m2. With the use of linear steps for scaling out and in, we could keep the minimum capacity as 0 and make sure that when the queue is empty, only 19 instances are removed, and one stays running. After making the aforementioned changes, we could scale in and out much more efficiently and process 100 requests in just 3 minutes and 27 seconds.

4. Code

4.1 App Tier

4.1.1 *app.py*

This Python script listens to an AWS Simple Queue Service (SQS) queue, receives messages containing names of images, downloads the images from an S3 bucket, processes the images by running a Python script for image classification, sends the results to another SQS queue and uploads the results to another S3 bucket. To begin with, the code imports the required modules, namely boto3, os, subprocess, and UUID. The boto3 module is the AWS SDK for Python, which provides a Pythonic interface to various AWS services such as S3, SQS, and many more. The os module is a Python module that uses operating system-dependent functionality like reading or writing to the file system. The subprocess module allows the script to run a separate Python script as a subprocess. Finally, the UUID module is used to generate unique identifiers.

The script then creates two clients, one for SQS and one for S3, with the region set to us-east-1. It then defines variables such as INPUT_IMAGE_DIR, request_queue_url, response_queue_url, input_bucket_name, and output_bucket_name. request_queue_url and response_queue_url are the URLs of the two SQS queues, which the script will use to send and receive messages. input_bucket_name and output_bucket_name are the names of the S3 buckets where the input and output images will be stored, respectively. It then creates a directory called input_images using the os module, which will be used to store the input images.

The script then enters a loop that listens to the message request queue. When a message is found, the input image name is extracted from the message and downloaded from the input S3 bucket using the s3.download_file() method. The image is then processed by running a Python script called image_classification.py using the subprocess.check_output() method, which returns the model's prediction. After obtaining the prediction, the downloaded image is deleted using the os.remove() method. The message is then deleted from the request queue using the sqs.delete_message() method. The image classification prediction is sent as a message into the response queue using the sqs.send_message() method. Finally, the prediction is uploaded to the output S3 bucket using the s3.put_object() method.

In conclusion, this script demonstrates using AWS services like SQS and S3 to implement an image processing pipeline. It uses the boto3 module to create clients to access these services, the os module to create directories and delete files, and the subprocess module to run a separate Python script for image classification.

4.2 Web Tier

4.2.1 *build.gradle*

This is a Gradle build file for a Java web application. The file declares a group, archivesBaseName, and version for the project. It also sets the source compatibility to Java 1.8 and specifies the Maven Central repository. It also declares several dependencies for the project, including the Spring Boot framework, Amazon Web Services SDK, J2HTML library, and Lombok library. Additionally, it sets up JUnit 5 for testing purposes.

4.2.2 *settings.gradle*

This is a gradle file responsible for setting the name of the project module of our java application. It will also be the name of the jar file generated on the project build unless specified otherwise by the build.gradle file.

4.2.3 *AWSProperties.java*

This file corresponds to a Java class named "AWSProperties", which represents a configuration class for Amazon Web Services (AWS) properties within a Spring Boot application. It features the "Data" annotation from the Lombok library, which enables the automatic generation of getter, setter, and other methods for class properties. Additionally, the "Value" annotation from the Spring framework allows the injection of property values from configuration files. This class encapsulates four private fields, representing the URLs for the SQS request and response queues and two S3 buckets for storing images and results. The fields are initialized with values from configuration files through the "@Value" annotation. The "@Component" annotation is also present, which marks the class as a Spring-managed bean for dependency injection. Overall, this class is a critical component in the AWS integration for the Spring Boot application.

4.2.4 *AWSSecrets.java*

This file corresponds to a Java class named "AWSSecrets" that contains a method for obtaining AWS credentials from the AWS Simple Systems Manager (SSM) parameter store. To interact with the AWS environment, the class imports the required AWS Java SDK packages, such as AWSSimpleSystemsManagement and BasicAWSCredentials. Additionally, the class imports the Properties class to process the parameter values returned by the SSM parameter store.The "getAWSCredentials" method retrieves the credentials stored in the SSM parameter store, encrypted with AWS Key Management Service (KMS). The technique uses the AWSSimpleSystemsManagementClientBuilder to create a client object for interacting with the SSM parameter store. It then executes a GetParameterRequest, which retrieves the parameter value stored in the specified SSM path. The parameter value is loaded into a Properties object to extract the access and secret keys. Finally, the method returns the extracted credentials in a BasicAWSCredentials object. This implementation of AWS credential retrieval is secure and scalable since it utilizes the SSM parameter store, which allows for centralized management and control of access to sensitive data. KMS encryption also ensures that the credentials are protected during storage and transmission. Implementing this class in a Spring Boot application using the "@Component" annotation enables dependency injection and promotes modularity and maintainability.

4.2.5 *WebTierRestController.java*

This Java file is a Rest Controller class that receives file uploads and returns the classification result of the uploaded image. It is annotated with @RestController and @RequestMapping("/"), indicating that this class handles RESTful web service requests and maps HTTP requests to handler methods. The class also has a single method handleFileUpload, that takes a file as a parameter and returns a string. This method is annotated with @PostMapping(value = "/file-upload"), indicating that it handles HTTP POST requests with "/file-upload" URI. The handleFileUpload method calls the handleFileUpload method of WebTierService class and returns the result. Using a MultipartFile parameter ensures that the uploaded file is received and processed correctly. This method is an appropriate way to handle file uploads and RESTful web service requests.

4.2.6 *WebTierService.java*

This class defines an interface called "WebTierService". It contains a method signature "handleFileUpload" that accepts a "MultipartFile" object as input and returns a "String" value. This method can throw two exceptions: "IOException" and "InterruptedException". This interface is a blueprint for a service that handles file uploads in a web tier. It defines what operations should be implemented by any class implementing this interface. The method "handleFileUpload" accepts a file as input, processes it, and returns a string value.

4.2.7 *WebTierServiceImpl.java*

This Spring Boot service class handles file uploads and interacts with AWS services, such as Amazon S3 and Amazon SQS. It provides two methods, receiveMessages() and handleFileUpload(). The receiveMessages() method is called during application startup and asynchronously polls the SQS queue for any new messages. The handleFileUpload() method takes a file and saves it to S3, sends a message containing the image name to the SQS request queue, waits for the response from the SQS response queue, and returns the result. It uses asynchronous programming using CompletableFuture to ensure that the application remains responsive while waiting for messages from the SQS response queue. Furthermore, it uses a ConcurrentHashMap to store results, allowing thread-safe and concurrent access to the resulting map. Finally, it uses the @Autowired annotation to automatically wire in the necessary dependencies, making the code cleaner and easier to read.

4.2.8 *S3Utility.java*

This file defines a utility class for interacting with Amazon S3 (Simple Storage Service) through the AWS SDK (Software Development Kit). It provides a method saveImageToS3() for uploading an image file to an S3 bucket specified in the application.properties. The method takes a String keyName as the file's name to be uploaded and a MultipartFile uploadedImageFile as the file to be uploaded. The method creates an ObjectMetadata object with the file's content length and content type, reads the file into an InputStream, and uses the AmazonS3 client object to upload the file to the specified bucket. The S3Utility class uses the Component annotation, which marks it as a Spring-managed bean that can be quickly injected into other classes. This promotes modular design and makes the code easier to maintain and test.

4.2.9 *SQSUtility.java*

This utility class handles Amazon Simple Queue Service (SQS) functionality. It provides methods for sending and receiving messages to and from an SQS queue. The SQS utility class has two methods, sendMsgToRequestQueue() and readMessages(), to send and read messages from the SQS queue, respectively. The class uses the AWS SDK to interact with SQS, which requires setting the AWS access key, secret key, and the AWS region. These configurations are provided through the AWSProperties class, which is injected into the SQSUtility class via Spring's dependency injection framework. The sendMsgToRequestQueue() method sends a message to the SQS request queue with the file/image name as the body of the message. The readMessages() method polls the SQS response queue and reads messages from it while storing it in a global map. The received messages are then parsed, and their values are added to the global map. The method continues to poll the queue until it is stopped manually. The code follows good programming practices, such as properly using the dependency injection framework and adhering to the SOLID principles.