CSE 546 --— Project Report

Sai Krishna Chilvery

Akash Reddy Maligireddy

Sadaf Shaik

1. Problem statement

Objective: Develop a scalable cloud application for image recognition using a provided deep learning model.

Core Requirements:

1. **Web Application:**

   Develop a web application that is capable of handling multiple concurrent requests using a queue mechanism (AWS SQS).

2. **Image Recognition:**

   Process user-uploaded images (.jpeg) and return the top-1 recognition result as plain text.

3. **Scalability and Concurrency:**

   Handle multiple concurrent requests and implement scaling between 1 to 20 instances, queuing additional requests. The scaling should be based on the depth of the amazon SQS Queue.

4. **Data Storage:**

   Persistently store input images and recognition results in separate S3 buckets, adhering to specified formats.

5. **Testing:**

   Utilize the provided `imagenet-100` dataset and workload generators for testing and validation.

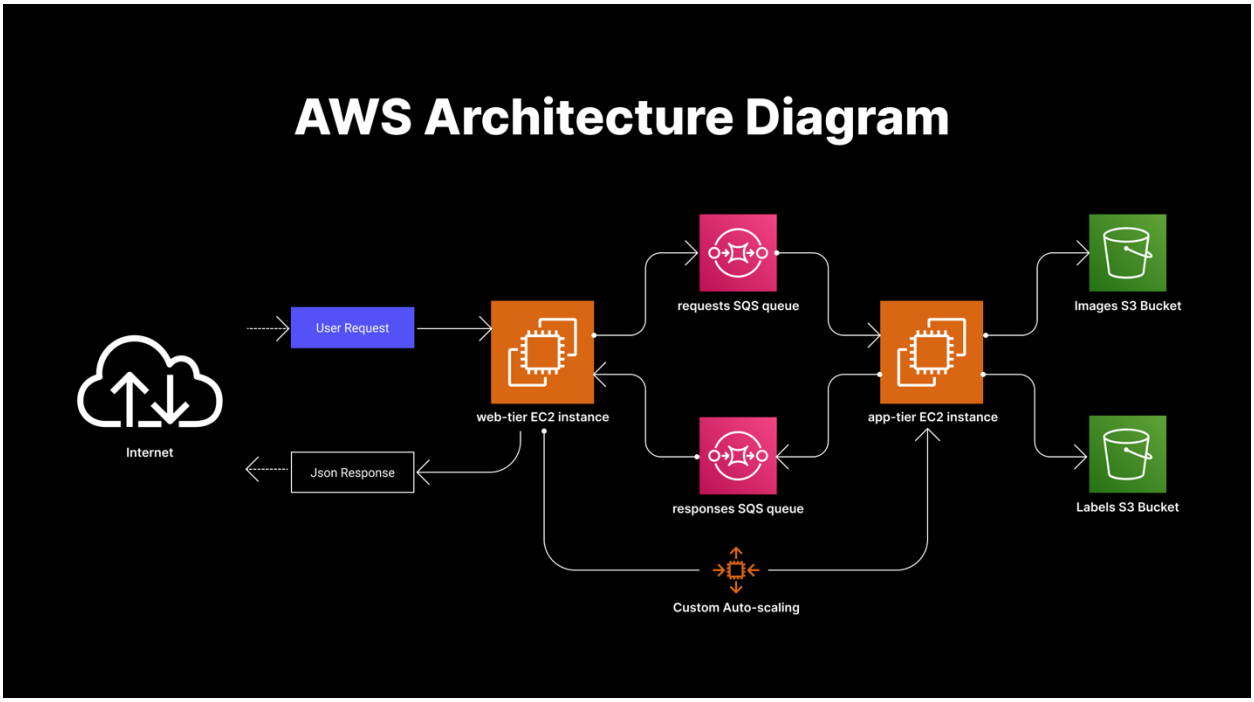
2. Design and implementation

2.1 Architecture

1. **Internet, User Request and JSON Response:**
   This section represents users and their interaction with the application. A typical user sends an image as a user request and receives label as a JSON Response.
2. **Web-Tier EC2 Instance**
   The web-tier EC2 Instance is the AWS compute platform instance we use to handle user requests. A brief working of the instance includes handling user requests and push them to the Requests SQS Queue.
3. **Custom Auto-Scaling**
   To handle auto-scaling, the web-tier controller monitors the requests queue depth and scales the app-tier instance linearly maxing out at 20 instances.
4. **Requests and Responses SQS Queues**

The request queue holds the user requests for processing. Using a queue helps in decoupling components, ensuring that the app-tier instance can handle the requests at its own pace and ensuring no request is lost even if there's a sudden surge. It also allows multiple app-tier instances to handle multiple requests in the queue.

5. **App-Tier EC2 Instance**
   This EC2 instance is dedicated to handling the image recognition tasks. Each app-tier instance subscribes to the responses queue and performs image recognition on the corresponding images. The app-tier instance also handles pushing images and their labels to the corresponding S3 buckets.
6. **Images and Labels S3 Buckets**
   The images and labels s3 buckets store the images received and labels obtained from the app-tier respectively. This ensures data-persistence for tasks like model training, data analysis etc., which are out of scope of this project currently.



2.2 Autoscaling

In the described architecture, a custom auto-scaling strategy is employed to ensure the system can handle varying workloads and respond dynamically to the demands.

The **web-tier** plays a crucial role in this scaling mechanism. While the web-tier receives and manages user requests, it actively monitors the depth of the **requests SQS queue** every **10 seconds**. The depth of this queue is a direct indication of the processing demand on the system, with a deeper queue indicating a higher number of pending tasks.

Based on the observed depth of the **requests SQS queue**, the web-tier triggers the scaling actions for the **app-tier**. Here's how the scaling logic functions:

1. **Scale-up**: As the depth of the queue increases, the web-tier signals for more app-tier EC2 instances to be provisioned. This is done linearly, meaning for a certain increase in queue depth, a proportional number of app-tier instances are added. This continues until a maximum of 20 app-tier instances are active. After reaching this threshold, even if the queue depth continues to grow, no more app-tier instances are added. This limitation ensures that there's controlled resource utilization and avoids potential pitfalls of unbounded scaling.
2. **Scale-down**: On the other hand, when the demand diminishes and the queue depth decreases, the web-tier triggers the de-provisioning of the app-tier instances. The system ensures that there's a linear reduction in the number of app-tier instances corresponding to the reduction in queue depth. However, the architecture always maintains a minimum of one app-tier instance active to ensure that there's always a capacity to process incoming requests, even if they are few.

In summary, the web-tier acts as the orchestrator for scaling operations, continuously gauging the processing demand by monitoring the **requests SQS queue** depth. Based on this real-time metric, it intelligently scales the app-tier instances up or down, ensuring that the system remains responsive during high demand periods and resource-efficient during quieter times.

2.3 Member Tasks

In the described project structure, each member has been assigned specific responsibilities relating to different components of the AWS architecture:

1. **Sai Krishna Chilvery - Web tier**: Sai Krishna Chilvery is responsible for the web-tier, which serves as the primary interface for user interactions. This entails:
    1. Designing, implementing, and managing the web servers or web services that receive user requests.
    2. Coordinating with other components, especially with the app-tier, by routing user requests appropriately, typically through the **requests SQS queue**.
    3. Constantly monitor the **responses SQS queue** and direct the responses to the corresponding users.
    4. Design the Web-tier AMI to be plug and play. This means, the entire application should be able to be deployed with just the web-tier AMI.
2. **Akash Reddy Maligireddy - Auto Scaling**: Akash Reddy Maligireddy is tasked with managing the auto-scaling mechanisms of the system. His responsibilities include:
    1. Work alongside web-tier development to implement auto-scaling.
    2. Monitoring the depth of the **requests SQS queue** at regular intervals (every 10 seconds) to make scaling decisions.
    3. Implementing the linear scaling logic, where the app-tier scales up or down in direct proportion to the queue depth, with specific constraints like capping at 20 instances and ensure that at least 1 **app-tier instance** is always available.
3. **Sadaf Shaik - App Tier**: Sadaf Shaik handles the app-tier, the backbone of the application's processing capabilities. Her role encompasses:

1. Build on the top of the provided Image recognition AMI. Consume the **requests SQS queue** to fetch tasks for processing and subsequently pushing results or responses to the **responses SQS queue**.
2. Integrating with storage solutions, like the **Images S3 Bucket** and **Labels S3 Bucket**. The consumed images are directed to the images bucket and the predicted labels to the labels bucket.
3. Serve App-tier as a one-shot AMI to ensure loose coupling with the web-tier instance and auto-scaling mechanisms.

The Testing and Validation of the project is done over a group discussion among the team. Together, these team members collaboratively ensure that the entire system, from the user-facing API to the backend processing and scaling mechanisms, operates seamlessly, efficiently, and responsively.

4. Testing and evaluation

Objective:

The primary objective of our project was to ascertain the system's capability to manage multiple concurrent requests. A key aspect of this evaluation was to compare the system's performance with and without the implementation of auto-scaling.

Testing Metrics:

The metric chosen for this evaluation was the time taken by the system to process a set number of concurrent requests. The results were analyzed both in scenarios with auto-scaling enabled and without auto-scaling.

Results:

The table below presents the recorded data:

Number of Requests	Without Auto-Scaling (seconds)	With Auto Scaling(seconds)	Average response time (without autoscaling)	Average response time (without autoscaling)
1	3.27	3.27	3.27	3.27
5	15.01	13.7	3.01	2.74
10	34.01	25.1	3.4	2.51
50	164.15	126.25	3.283	2.51
100	325.91	180.53	3.259	1.8

Analysis:

From the data, it's evident that auto-scaling significantly improves the system's response time as the number of concurrent requests increases. While the advantages of auto-scaling are subtle with lower request numbers, the benefits become pronounced as the request volume escalates.

4. Code

Web-Tier

1. ***Response Handler (web-tier/response_handler.py)*:**

The `ResponseHandler` class in response_handler.py provides a simple interface for storing and retrieving responses to messages. The class uses an SQLite database to store the responses, with each response associated with a unique `messageId`. The class provides methods for adding a response to the database, deleting a response from the database, and retrieving a response from the database.

The `__init__` method of the `ResponseHandler` class initializes a connection to the SQLite database and creates the responses table if it does not already exist.
The `_create_responses_table` method creates the responses table with two columns: `messageId` and `response`. The `messageId` column is defined as the primary key, which ensures that each `messageId` is unique and can only be associated with one response.

The `_delete_response` method of the `ResponseHandler` class deletes a response from the responses table with the specified `messageId`. This method is called after a response has been retrieved from the table, to ensure that the response is not retrieved again in the future.

The `get_response method` of the `ResponseHandler` class retrieves a response from the responses table with the specified `messageId`. The method waits for a maximum of `max_timeout` seconds for a response to become available, polling the table every `0.5 seconds`. If a response is retrieved from the table, the response is deleted from the table using the `_delete_response` method. If no response is retrieved within the specified timeout, the method returns None.

Overall, the `ResponseHandler` class provides a simple and efficient way to store and retrieve responses to messages using an SQLite database. The class can be used in a wide range of applications, such as chatbots, web applications, and more. To use the class, simply create an instance of the `ResponseHandler` class and call the `add_response` and `get_response` methods as needed.

2. ***SQS Queue Manager (web-tier/queue_manager.py)***

The `SQSQueueManager` class in queue_manager.py provides a simple interface for managing Amazon Simple Queue Service (SQS) queues and Amazon Simple Storage Service (S3) buckets. The class uses the boto3 library to interact with the AWS services.

The __init__ method of the SQSQueueManager class initializes the sqs and s3 clients, sets up the request and response queues, and creates the ResponseHandler and InstanceController objects.

The `consumer_greenlet` and `monitor_greenlet` greenlets are spawned to handle consuming messages from the response queue and monitoring the request queue size, respectively. The setup method of the `SQSQueueManager` class sets up the request and response queues and the S3 buckets if they do not already exist. The `_setup_queue` method creates a queue with the specified name if it does not already exist and returns the URL of the queue.

The `_setup_s3_bucket` method creates a bucket with the specified name if it does not already exist.

The `handle_request` method of the `SQSQueueManager` class sends a message to the request queue with the specified image and filename. The image is read and base64-encoded before being sent as the message body. The `response_handler` object is used to add a response with the message ID of the sent message.

The `get_result` method of the `SQSQueueManager` class retrieves the response for the specified message ID from the `response_handler` object. If a response is not yet available, the method returns None. If a response is available, the method returns the result string, which is the second comma-separated value in the response.

The consume method of the `SQSQueueManager` class consumes messages from the response queue and adds them to the `response_handler` object. The method polls the queue every 20 seconds and can receive up to 10 messages at a time. The `OriginMessage` message attribute is used to associate the response with the original request message.

The `_get_request_queue_size` method of the `SQSQueueManager` class retrieves the approximate number of messages in the request queue. The monitor method of the `SQSQueueManager` class monitors the request queue size and scales the number of instances in the `InstanceController` object accordingly. If the queue is empty, the method scales the number of instances to 1. If the queue has fewer messages than the maximum number of instances, the method scales the number of instances to the number of messages. If the queue has more messages than the maximum number of instances, the method scales the number of instances to the maximum number of instances.

Overall, the `SQSQueueManager` class provides a simple and efficient way to manage SQS queues and S3 buckets in AWS. The class can be used in a wide range of applications, such as image processing, data analysis, and more. To use the class, simply create an instance of the `SQSQueueManager` class and call the `handle_request` and `get_result` methods as needed. The class takes care of sending and receiving messages and scaling the number of instances based on the queue size.

3. Server (web-tier/server.py)

The server.py file contains the implementation of a Flask web server that provides an API for uploading images and retrieving the results of image processing tasks. The server uses the `SQSQueueManager` class from queue_manager.py to manage the Amazon Simple Queue Service (SQS) queues and Amazon Simple Storage Service (S3) buckets used for processing the images.

The app object is an instance of the Flask class, which is used to create the web server. The `queue_manager` object is an instance of the `SQSQueueManager` class, which is used to manage the SQS queues and S3 buckets.

The `upload_image` function is the main endpoint for uploading images. It expects a POST request with a file named myfile in the request body. If no file is provided, the function returns a JSON response with an error message and a 400 status code. If a file is provided, the function passes the file and its filename to the handle_request method of the `queue_manager` object to add the request to the request queue. The function then calls the get_result method of the `queue_manager` object to retrieve the result of the processing task associated with the message ID returned by the handle_request method. If the result is not yet available, the function returns a 400 status code. If the result is available, the function returns a JSON response with the result.

The if `__name__ == '__main__'`: block starts the Flask web server on port 80 and listens for incoming requests. The debug parameter is set to False to disable debug mode.

Overall, the server.py file provides a simple and efficient way to upload images and retrieve the results of image processing tasks using a Flask web server and the `SQSQueueManager` class. The server can be used in a wide range of applications, such as image recognition, object detection, and more. To use the server, simply run the server.py file and send POST requests to the / endpoint with an image file in the request body.

Auto-Scaling

1. Instance Controller (web-tier/instance_controller.py)

The `InstanceController` class in instance_controller.py provides a simple interface for managing Amazon Elastic Compute Cloud (EC2) instances. The class uses the boto3 library to interact with the AWS services.

The __init__ method of the `InstanceController` class initializes the ec2 resource and sets up the instance names and instances dictionary. The `setUp` method sets up the instance names and instances dictionary and scales the number of instances to 1.

The `_generate_instance_names` method of the `InstanceController` class generates a dictionary of instance names with the format `app-instance-i`, where i is a number between 1 and the maximum number of instances.

The _get_instance_ec2 method of the `InstanceController` class retrieves the EC2 instance object with the specified name if it exists and is running.

The `_get_instances` method of the `InstanceController` class retrieves the EC2 instance objects for all instance names and updates the instances dictionary.

The `_get_first_available_instance_name` method of the `InstanceController` class retrieves the name of the first available instance, if any.

The `_get_latest_instance_name` method of the `InstanceController` class retrieves the name of the latest launched instance, if any.

The `create_instance` method of the `InstanceController` class creates a new EC2 instance with the specified name and configuration.

The `terminate_instance` method of the `InstanceController` class terminates the latest launched instance, if any.

The `get_live_instances` method of the `InstanceController` class retrieves the names of all running instances.

The `scale_to_count` method of the `InstanceController` class scales the number of instances to the specified count. If the count is less than 1 or greater than the maximum number of instances, the method returns an error message. If the current number of instances is less than the specified count, the method creates new instances until the count is reached. If the current number of instances is greater than the specified count, the method terminates the latest launched instances until the count is reached.

Overall, the `InstanceController` class provides a simple and efficient way to manage EC2 instances in AWS. The class can be used in a wide range of applications, such as web servers, data processing, and more. To use the class, simply create an instance of the `InstanceController` class and call the `create_instance`, `terminate_instance`, and `scale_to_count` methods as needed. The class takes care of launching and terminating instances and scaling the number of instances based on the specified count.

App-Tier

1. ***Response Handler (web-tier/response_handler.py)*:**

   The consumer.py file contains the implementation of a consumer that processes messages from an Amazon Simple Queue Service (SQS) queue and performs image classification on the images. The consumer uses the boto3 library to interact with the AWS services.

   The `Consumer` class is the main class that implements the consumer. The `__init__` method of the `Consumer` class initializes the SQS and S3 clients and sets up the request and response queue URLs.

   The setup method of the Consumer class sets up the request and response queue URLs by calling the `_get_queue_url` method with the queue names.

   The `_get_queue_url` method of the Consumer class retrieves the URL of the specified queue name, if it exists. If the queue does not exist, the method raises an exception.

   The `_put_img_s3` method of the Consumer class uploads the image to the specified S3 bucket with the specified filename using the `s3.put_object` method.

   The `_put_result_s3` method of the Consumer class uploads the result of the image classification to the specified S3 bucket with the image name as the key using the s3.put_object method.

   The `_process_image` method of the Consumer class performs image classification on the specified image file using the image_classification.py script. The method saves the image to a local file, runs the image recognition script using `subprocess.run`, and retrieves the output of the script. The method then extracts the label from the output and returns the image name and label as a string.

   The consume method of the Consumer class is the main method that processes messages from the request queue. The method retrieves messages from the request queue using the `sqs.receive_message` method, processes each message using

the `_put_img_s3`, `_process_image`, and `_put_result_s3` methods, and sends the result to the response queue using the produce method. The method then deletes the message from the request queue using the `sqs.delete_message` method.

The produce method of the Consumer class sends the result of the image classification to the response queue with the message ID of the original request message as a message attribute using the `sqs.send_message` method.

Overall, the consumer.py file provides a simple and efficient way to process messages from an SQS queue and perform image classification on the images. The consumer can be used in a wide range of applications, such as image recognition, object detection, and more. To use the consumer, simply create an instance of the Consumer class and call the consume method. The class takes care of retrieving messages from the request queue, performing image classification, and sending the result to the response queue.

Contributions - Sai Krishna Chilvery

Contributions: Web-tier

- **Setup AWS**: Spearheaded the initial configuration on AWS, ensuring that the infrastructure was fine-tuned to meet the project's specific requirements.
- **Initial Resources Setup**: Paved the groundwork for the project by setting up crucial AWS resources. This included configuring EC2 instances, designing appropriate IAM roles, and ensuring that the necessary permissions were granted.
- **Response Handler**: Crafted a sophisticated response management mechanism that ensures timely feedback from the app-tier to users, enhancing user experience and interaction.
- **Queue Manager**: Harnessing the power of AWS's SQS service, designed a robust queue system that efficiently manages incoming user requests. This design ensures a seamless and resilient interaction between the web-tier and app-tier, optimizing response times.
- **Create Server**: Led the establishment of the primary server for the web-tier, ensuring its optimal configuration to adeptly handle, process, and route incoming requests.
- **Copy Files to Server**: Undertook the crucial task of transferring project files, scripts, and other essential assets to the server, ensuring the server's readiness for optimal performance.
- **Serve with Gunicorn**: Integrated the web application with the Gunicorn 'Green Unicorn' server, a strategic move that guarantees the system's ability to handle multiple concurrent requests efficiently.

Skills Learned:

- **AWS python boto3 client**: Acquired proficiency with the boto3 client, enabling nuanced interactions and automations with AWS services.
- **Concurrent Programming**: Gained in-depth knowledge of concurrent programming principles, equipping myself to efficiently manage simultaneous user interactions.
- **SQS Queues**: Deepened understanding of Amazon's SQS service and its capabilities, becoming adept at crafting resilient queuing systems for efficient request management.
- **Publish-subscribe Model**: Familiarized with the pub-sub communication model, recognizing its potential to augment system flexibility and scalability through asynchronous interactions.
- **AWS EC2**: Developed expertise in utilizing Amazon Elastic Compute Cloud (EC2), understanding its core features and functionalities for deploying and managing virtual servers in the cloud.
- **AWS S3**: Gained comprehensive knowledge of Amazon Simple Storage Service (S3), learning how to store, retrieve, and protect data in scalable cloud storage.
- **AWS CLI**: Gained hands-on experience with the AWS Command Line Interface (CLI), equipping myself with the skills to manage AWS services directly from the terminal.

Contributions - Akash Reddy Maligireddy

Contributions: Auto Scaling

- **Collaboration with Web-tier**: Worked closely with the web-tier development team to seamlessly integrate and implement auto-scaling mechanisms that support the architecture's responsiveness and resilience.
- **Monitoring and Decision-making**: Proactively monitored the depth of the requests SQS queue at regular intervals, specifically every 10 seconds. This rigorous monitoring aids in making informed scaling decisions to maintain system efficiency.
- **Scaling Logic Design**: Spearheaded the creation and implementation of a robust linear scaling logic, ensuring the app-tier resources scale in harmony with queue depth variations.
- **Resource Optimization**: Implemented strategic constraints in the scaling logic, such as a maximum cap of 20 instances for scalability. This not only optimized resource usage but also ensured cost-effectiveness.
- **System Reliability**: Introduced safety protocols guaranteeing that at least one app-tier instance always remains operational, a pivotal step for maintaining uninterrupted service.
- **Real-time Adjustments**: Developed mechanisms to adapt to sudden surges or drops in traffic, ensuring that the system remains responsive and efficient even during unexpected scenarios.

Skills Learned:

- **AWS SQS Queues**: Acquired a deep understanding of Amazon's SQS service, enabling proficient management of messages in a scalable and decoupled manner.
- **AWS**: Gained comprehensive knowledge of AWS's suite of services and tools, understanding their intricacies and how they can be harnessed to build scalable solutions.
- **Python boto3 client**: Developed expertise in using the boto3 client for Python, facilitating streamlined interactions and efficient automations with AWS services.
- **Auto-Scaling Expertise**: Deepened understanding of auto-scaling principles and AWS's native tools that support the creation of responsive and resilient architectures.
- **Queue Management**: Enhanced skills in queue management, specifically leveraging SQS metrics to make informed scaling decisions.
- **Scalability Logic**: Honed the ability to design algorithms that dictate resource scaling, ensuring optimal utilization and consistent system performance.

Contributions - Sadaf Shaik

Contributions: App Tier

- **Integration with AMI**: Leveraged the foundational Image recognition AMI to enhance the core processing capabilities of the app-tier. This foundation has been pivotal in achieving seamless processing and response mechanisms.
- **SQS Management and Integration**: Expertly subscribed to the requests SQS queue to consume tasks and subsequently publish results to the responses SQS queue after processing. This ensures a smooth and efficient data flow between the app-tier and other system components.
- **S3 Buckets Management**: Skillfully integrated with AWS storage solutions. Managed the direction of consumed images to the Images S3 Bucket while directing accurate prediction results to the Labels S3 Bucket.
- **App-tier Independence**: Conceived and implemented the App-tier as a one-shot AMI, providing it with a level of independence from the web-tier instance. This strategic move not only ensured system resilience but also optimized interactions with auto-scaling mechanisms.

Skills Learned:

- **Amazon Machine Images (AMI)**: Delved deep into the complexities of AMIs, honing skills particularly around image recognition and processing.
- **AWS SQS Mastery**: Acquired a nuanced understanding of Amazon's Simple Queue Service. Developed proficiency in subscribing to and publishing messages, ensuring a seamless and efficient data flow within the application.
- **S3 Bucket Management**: Gained comprehensive hands-on experience with Amazon's Simple Storage Service (S3), mastering the nuances of cloud storage management.
- **System Architecture Design**: Cultivated an in-depth understanding of designing and implementing loosely coupled systems that ensure scalability, resilience, and efficient data flow in cloud environments.