**Interview Question**

**Question 1**

**S3 versioning ?**

Amazon S3 (Simple Storage Service) provides a feature called "versioning," which allows you to keep multiple versions of an object in a bucket. This feature helps protect against unintended overwrites and deletions, as well as provides an audit trail of changes made to objects over time. Here's how versioning works in Amazon S3:

### How S3 Versioning Works:

1. **Enabling Versioning**:
   * Versioning can be enabled at the bucket level.
   * Once enabled, S3 stores a new version of an object each time you upload it, even if it has the same name as an existing object.
   * All subsequent uploads or updates to the same key result in additional versions of the object.
2. **Unique Version IDs**:
   * Each version of an object in S3 is assigned a unique version ID.
   * Version IDs are hexadecimal strings, such as **"3/L4kqtJlcpXroDTDmpUMLUo"**.
3. **Listing Object Versions**:
   * You can list all versions of an object using the **ListObjectVersions** API or the AWS Management Console.
   * Each object in S3 has a special "current" version, which is the version returned by default when you perform a GET operation on the object without specifying a version ID.
4. **Deleting Object Versions**:
   * Deleting an object in a versioned bucket does not actually delete the object. Instead, it adds a delete marker to indicate that the object is deleted.
   * You can delete specific versions of an object by specifying the version ID, or you can delete all versions of an object, including delete markers, to permanently remove it.
5. **Accessing Object Versions**:
   * You can retrieve a specific version of an object by specifying its version ID in the GET request.
   * If you don't specify a version ID, you'll receive the current version of the object.
6. **Cost Implications**:
   * Storing multiple versions of objects can increase storage costs, especially if objects are frequently updated or have large sizes.
   * You are charged for the storage of all object versions, including delete markers.

### Basic Operations with Versioning:

1. **Enabling Versioning**:
   * Versioning can be enabled or suspended using the AWS Management Console, AWS CLI, or AWS SDKs.
2. **Uploading Objects**:
   * Objects are uploaded to versioned buckets as usual, using the **PutObject** API or similar methods.
   * Each upload creates a new version of the object, unless the uploaded object is identical to the current version.
3. **Listing Object Versions**:
   * Use the **ListObjectVersions** API or the AWS Management Console to list all versions of an object in a versioned bucket.
4. **Deleting Objects**:
   * Deleting an object adds a delete marker, which indicates that the object is logically deleted.
   * To permanently delete an object, you must delete all versions, including delete markers, associated with the object.
5. **Retrieving Object Versions**:
   * You can retrieve specific versions of an object by specifying their version IDs in GET requests.
   * If you don't specify a version ID, you'll receive the current version of the object.

### Example (Using AWS CLI):

Here's an example of how you can enable versioning for an S3 bucket using the AWS CLI:

bashCopy code

aws s3api pt-bucket-versioning --bucket your-bucket-name --versioning-configuration Status=Enabled

Replace **your-bucket-name** with the name of your S3 bucket.

### Considerations:

* Be mindful of storage costs when using versioning, especially if your objects are large or frequently updated.
* Versioning provides protection against accidental data loss but does not protect against intentional overwrites or deletions.
* Use versioning in conjunction with other AWS features, such as MFA delete, to add additional layers of security and control.

**Question 2**

**Event driven solution by sns, s3 and lambda ?**

An event-driven solution using Amazon S3, Amazon SNS (Simple Notification Service), and AWS Lambda allows you to automate workflows based on events occurring in your S3 bucket. Here's how you can set up such a solution:

### Scenario:

Let's consider a scenario where you want to trigger a notification (such as an email) whenever a new object is uploaded to a specific folder in your S3 bucket.

### Solution Architecture:

1. **S3 Bucket**:
   * Create an Amazon S3 bucket where you'll upload your objects.
   * Configure event notifications on the bucket to trigger an AWS Lambda function whenever a new object is uploaded to a specific folder.
2. **Lambda Function**:
   * Create an AWS Lambda function that will be triggered by S3 events.
   * Configure the Lambda function to process the event, such as sending a notification using Amazon SNS.
3. **SNS Topic**:
   * Create an Amazon SNS topic to which the Lambda function will publish notifications.
   * Configure email subscriptions to the SNS topic to receive notifications.

### Implementation Steps:

1. **Create an S3 Bucket**:
   * Create an S3 bucket in the AWS Management Console if you haven't already.
2. **Configure Bucket Events**:
   * Go to the S3 bucket's properties and select "Events".
   * Click "Add notification" and configure an event notification for the desired event (e.g., "All object create events") on the desired folder within the bucket.
   * Choose the Lambda function you created as the target for the event.
3. **Create a Lambda Function**:
   * Write a Lambda function in your preferred programming language that will be triggered by S3 events.
   * The function should process the event payload to extract information about the uploaded object (e.g., bucket name, object key).
   * Use the AWS SDK to publish a message to an SNS topic with the relevant notification information.
4. **Create an SNS Topic**:
   * Go to the SNS service in the AWS Management Console and create a new topic.
   * Add email subscriptions to the topic for the recipients who should receive notifications.
5. **Subscribe Lambda Function to SNS Topic**:
   * In your Lambda function code, subscribe the Lambda function to the SNS topic to publish notifications.
   * Ensure the Lambda function has permissions to publish messages to the SNS topic.
6. **Test the Solution**:
   * Upload a file to the specified folder in the S3 bucket and verify that a notification is sent to the subscribed email addresses.

**Question 3**

**lambda will get trigger whenever there is message in queue ? explain with example ?**

Yes, you can trigger an AWS Lambda function whenever there is a new message available in an Amazon SQS (Simple Queue Service) queue. This is achieved by configuring the SQS queue as an event source for the Lambda function. When a new message is added to the queue, it triggers the Lambda function, allowing you to process the message in a serverless manner.

### Example Scenario:

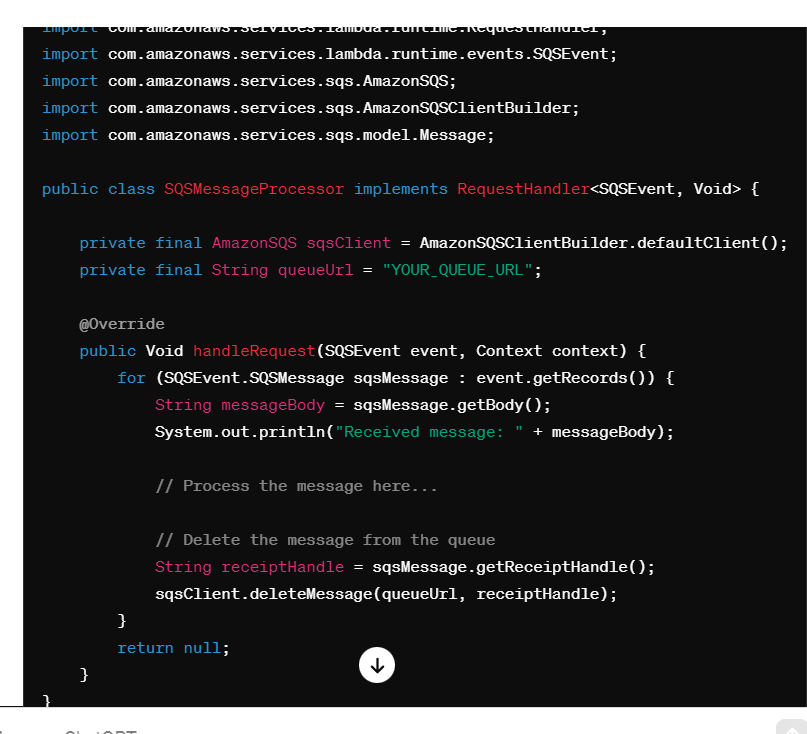
Let's consider a scenario where you have an SQS queue named "MyQueue" and you want to trigger a Lambda function whenever a new message is added to this queue. The Lambda function will process the incoming messages and perform some task, such as sending an email notification.

### Implementation Steps:

1. **Create an SQS Queue**:
   * Go to the SQS service in the AWS Management Console and create a new queue named "MyQueue".
2. **Create a Lambda Function**:
   * Write a Lambda function in your preferred programming language that will be triggered by SQS events.
   * This function will process the messages received from the SQS queue.
3. **Configure SQS as Event Source for Lambda**:
   * Go to the Lambda service in the AWS Management Console and select your Lambda function.
   * Add an event source and select "SQS" as the trigger type.
   * Choose the SQS queue "MyQueue" as the event source.
   * Configure the batch size and other settings as needed.
4. **Write Lambda Function Code**:
   * Write the code for your Lambda function to process the incoming messages from the SQS queue.
   * The function should retrieve the messages from the SQS queue and perform the necessary processing.
5. **Test the Solution**:
   * Upload a message to the "MyQueue" SQS queue and verify that the Lambda function is triggered and processes the message.

### Example Lambda Function (Java):

Here's a simple example of how you might implement the Lambda function in Java:

****

In this example, the Lambda function processes the incoming SQS messages by retrieving their body, printing them to the console, and then deleting them from the queue to avoid reprocessing. Make sure to replace **"YOUR\_QUEUE\_URL"** with the URL of your SQS queue.

Once the Lambda function is configured to listen to the SQS queue, it will automatically be triggered whenever a new message is added to the queue.

**Question 4**

**unable to access the message in the queue in aws what will happen ?**

If your AWS Lambda function is unable to access the message in the SQS (Simple Queue Service) queue, a few different scenarios could occur:

1. **Lambda Function Errors**:
   * If your Lambda function encounters an error while attempting to access the message in the SQS queue, it may fail to process the message.
   * Depending on your Lambda function's error handling configuration, the failure might result in retries, invocation errors, or dead-letter handling.
2. **Visibility Timeout Expiry**:
   * When a message is retrieved from an SQS queue by a Lambda function, it becomes temporarily invisible to other consumers for a specified visibility timeout period.
   * If your Lambda function fails to process the message within this visibility timeout period, the message becomes visible in the queue again, allowing other consumers (or the same Lambda function upon retry) to retrieve it.
3. **Lambda Function Invocation Limits**:
   * AWS Lambda has limits on the number of concurrent executions and the maximum execution duration.
   * If your Lambda function is throttled or reaches its concurrency limit, it may not be able to process messages from the SQS queue in a timely manner.
4. **Permissions Issues**:
   * Ensure that your Lambda function has the necessary permissions to access the SQS queue.
   * Check that the Lambda execution role includes the **SQS:ReceiveMessage** permission for the specified queue.
5. **Message Processing Logic**:
   * Review your Lambda function's code to ensure that it correctly handles SQS messages, including deserialization of message bodies and proper error handling.
   * Verify that the message format expected by your Lambda function matches the format of messages in the SQS queue.
6. **Queue Configuration**:
   * Check the configuration of your SQS queue, including visibility timeout, message retention period, and other settings, to ensure they align with your use case requirements.
7. **Network Connectivity Issues**:
   * Ensure that your Lambda function and SQS queue are deployed in the same AWS region to avoid network latency or connectivity issues.
   * Monitor for any network-related errors or issues that may affect communication between Lambda and SQS.

To diagnose and troubleshoot issues with your Lambda function's access to messages in the SQS queue, you can:

* Review CloudWatch logs for your Lambda function to identify any error messages or exceptions.
* Check the Lambda function's invocation metrics in the AWS Management Console to see if it's processing messages efficiently.
* Examine SQS queue metrics to monitor message delivery and visibility timeout behavior.
* Use AWS X-Ray for distributed tracing to analyze the performance and execution flow of your Lambda function and its interactions with the SQS queue.

**Question 5 :-**

sqs basic scope ? how to add item into queue ?

The basic scope of Amazon SQS (Simple Queue Service) involves creating a queue, adding items (messages) to the queue, and processing those items. Let's break down each step:

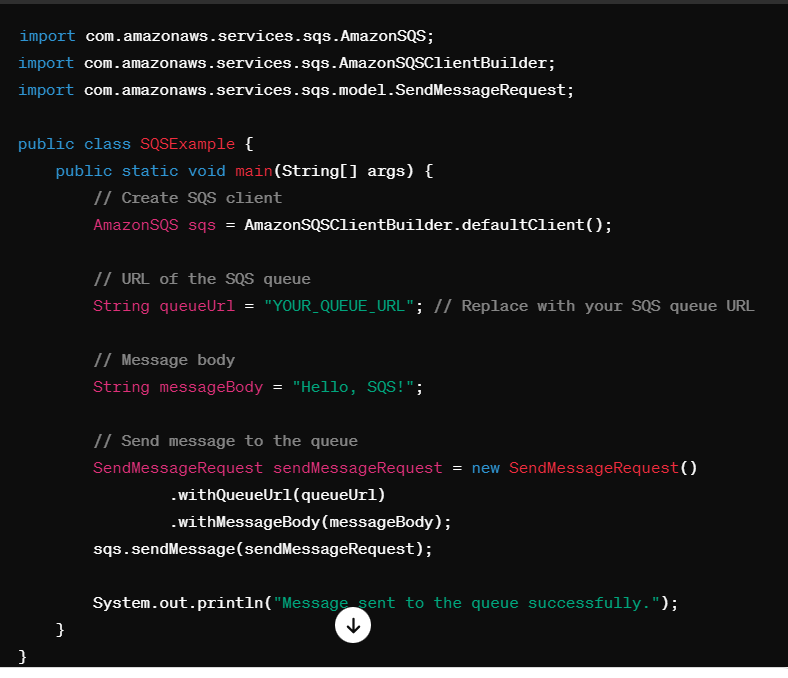
### 1. Create an SQS Queue:

* Go to the SQS service in the AWS Management Console.
* Click on "Create queue" and provide a name for your queue.
* Optionally, configure queue settings such as message retention period, visibility timeout, and others.
* Click on "Create queue" to create the queue.

### 2. Add Items (Messages) to the Queue:

* You can add messages to an SQS queue programmatically using the AWS SDK for the programming language of your choice (e.g., Java, Python, Node.js).
* Each message can be up to 256 KB in size and can contain any information you want to send to the queue.
* Messages are added to the queue using the **SendMessage** API operation or equivalent method in the SDK.

#### Example (Using AWS SDK for Java):

****

### . Process Items in the Queue:

* You can have one or more consumers (such as AWS Lambda functions or EC2 instances) retrieve messages from the queue and process them.
* Messages are retrieved from the queue using the **ReceiveMessage** API operation or equivalent method in the SDK.
* After processing a message, the consumer should delete the message from the queue to remove it.

#### Example (Using AWS SDK for Java to Receive and Delete Messages):

****

Ensure you have the necessary IAM permissions set up for your AWS credentials to interact with SQS, including permissions to send messages to the queue and receive/delete messages from the queue. Additionally, replace **"YOUR\_QUEUE\_URL"** with the actual URL of your SQS queue.

Question 6

**session and config to connect with aws ?**

To connect to AWS services programmatically using the AWS SDK for Java, you typically need to provide credentials and specify the AWS region. Here's how you can set up the AWS SDK for Java to connect to AWS using session and configuration:

### 1. AWS Credentials:

#### Option 1: Environment Variables:

You can set the following environment variables with your AWS credentials:

* **AWS\_ACCESS\_KEY\_ID**: Your AWS access key ID.
* **AWS\_SECRET\_ACCESS\_KEY**: Your AWS secret access key.
* **AWS\_SESSION\_TOKEN** (optional): Your AWS session token if using temporary security credentials.

#### Option 2: AWS Credentials Provider Chain:

Alternatively, you can use the AWS credentials provider chain, which automatically looks for credentials in various locations, including environment variables, system properties, the AWS credentials file, and IAM instance profile credentials.

### 2. AWS Region:

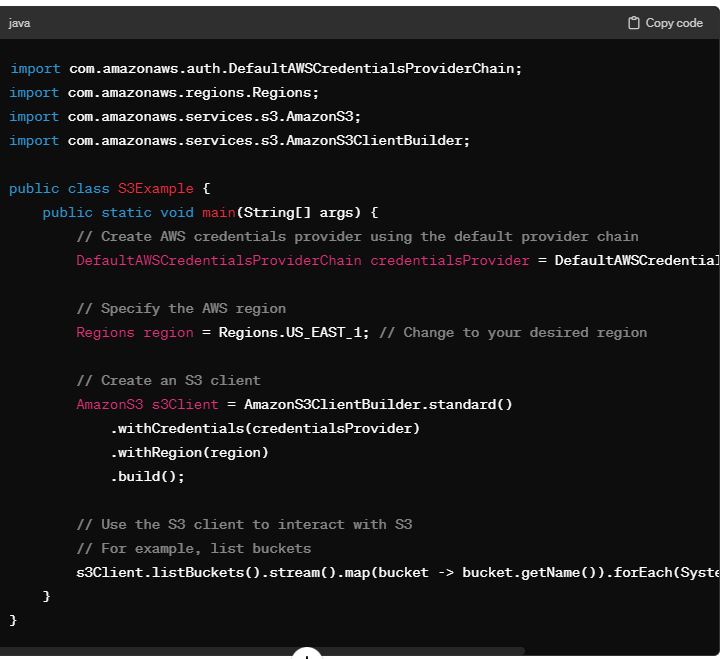
#### Option 1: Environment Variable:

Set the **AWS\_REGION** environment variable to specify the AWS region to use.

#### Option 2: Programmatically:

You can specify the region programmatically in your code.

### Example Code:

****

Ensure you have the necessary IAM permissions set up for your AWS credentials to access the AWS services you're using (e.g., Amazon S3). This example demonstrates connecting to Amazon S3, but you can similarly connect to other AWS services using their respective client builders (e.g., **AmazonDynamoDBClientBuilder** for Amazon DynamoDB).

Replace **Regions.US\_EAST\_1** with the desired AWS region and customize the AWS service client according to your requirements.

**Question 7**

**Dynamo db rate**

In Amazon DynamoDB, the term "rate" can refer to different aspects depending on the context. Here are a few interpretations:

### 1. Provisioned Capacity Rate:

* DynamoDB allows you to provision read and write capacity for your tables based on your expected workload.
* The provisioned capacity rate refers to the amount of read/write throughput capacity provisioned for your DynamoDB table.
* You can specify the provisioned capacity in terms of read capacity units (RCUs) and write capacity units (WCUs).
* One read capacity unit (RCU) represents one strongly consistent read per second for items up to 4 KB in size, or two eventually consistent reads per second.
* One write capacity unit (WCU) represents one write per second for items up to 1 KB in size.
* You can adjust the provisioned capacity rate for your tables based on your application's needs.

### 2. Request Rate:

* The request rate refers to the rate at which your application makes requests to DynamoDB.
* DynamoDB charges for read and write requests in addition to provisioned capacity.
* Read requests are charged based on the number of read capacity units (RCUs) consumed, and write requests are charged based on the number of write capacity units (WCUs) consumed.
* DynamoDB also imposes limits on the number of read and write requests per second (per table/partition), which can affect the request rate.

### 3. Throttling Rate:

* DynamoDB employs throttling mechanisms to ensure that a single table/partition doesn't consume too much provisioned capacity or overwhelm the system.
* If your application exceeds the provisioned capacity or request limits, DynamoDB may throttle requests by returning **ProvisionedThroughputExceededException** or **ThrottlingException**.
* Throttling can occur when the request rate exceeds the provisioned capacity rate or when burst capacity limits are exceeded.

### Managing Rate and Performance:

* Monitor your DynamoDB tables' provisioned capacity utilization and adjust capacity as needed to handle changes in workload.
* Design your application to handle DynamoDB throttling gracefully by implementing retry logic with exponential backoff.
* Consider using DynamoDB on-demand capacity mode, which automatically scales read and write capacity based on the workload without requiring provisioned capacity configuration.

Understanding and effectively managing the rate at which your application interacts with DynamoDB, both in terms of provisioned capacity and request rate, is crucial for achieving optimal performance and cost-effectiveness.

**Question 8**

**What happen in background when we push code in github ?**

When you push code to a repository hosted on GitHub, several background processes occur:

1. **Authentication and Authorization**:
   * GitHub verifies your identity and permissions to access the repository based on your credentials (username/password or SSH key).
   * Access control mechanisms ensure that only authorized users can push code to the repository.
2. **Data Transfer**:
   * Your local Git client sends the new commits and changes to the remote repository on GitHub.
   * This involves transferring data over the internet using HTTPS or SSH protocols, depending on how the repository is configured.
3. **Validation and Processing**:
   * GitHub validates the received data to ensure it conforms to Git protocol specifications.
   * It checks for any conflicts or errors in the pushed code and verifies that it doesn't violate repository rules (e.g., branch protection rules, file size limits).
4. **Branch Update**:
   * If you're pushing changes to an existing branch, GitHub updates the branch with the new commits and changes.
   * If you're pushing changes to a new branch, GitHub creates the branch in the repository.
5. **Hooks and Automation**:
   * GitHub triggers any configured webhooks, continuous integration (CI) pipelines, or automation workflows associated with the repository.
   * This allows for automated testing, code analysis, deployment, or notifications based on the pushed code changes.
6. **Notification and Collaboration**:
   * GitHub notifies collaborators and subscribers of the repository about the pushed changes.
   * Collaborators can review the changes, leave comments, or initiate discussions using pull requests or issue comments.
7. **Persistence and Backup**:
   * The pushed code changes are persisted in the repository's Git database, ensuring durability and data integrity.
   * GitHub takes regular backups and maintains redundant copies of repositories to prevent data loss.
8. **Access Control and Audit Logging**:
   * GitHub logs all push events and actions performed by users for auditing and compliance purposes.
   * Access control mechanisms ensure that only authorized users can view or modify the repository's contents.

**Question 9:**

any retrigger mechanism to get message from queue ?

Yes, there is a mechanism to retrigger the retrieval of messages from an Amazon SQS (Simple Queue Service) queue in case the initial retrieval attempt fails or the messages are not processed successfully. This mechanism involves the concept of message visibility timeout and the use of long polling or retries.

### 1. Message Visibility Timeout:

* When a consumer retrieves a message from an SQS queue, the message becomes invisible to other consumers for a specified duration known as the visibility timeout.
* If the message is not deleted (or becomes visible again due to the timeout expiry), it becomes available for retrieval by other consumers or the same consumer for retries.

### 2. Long Polling:

* Long polling is a mechanism that allows a consumer to wait for messages in the queue for a specified period (up to 20 seconds) before returning a response.
* When a consumer polls the queue with a long polling request, the request remains open until a message is available or the long polling timeout expires.
* Long polling reduces the number of empty responses returned when the queue is empty and helps conserve resources.

### 3. Message Retries:

* If a consumer fails to process a message successfully (e.g., due to transient errors or processing failures), it can retry processing the message after a certain delay.
* The delay between retries can be gradually increased using exponential backoff to prevent overwhelming the system with retry attempts.
* If the message processing still fails after multiple retries, the consumer can return the message to the queue for further processing by other consumers or future retries.

### Example Scenario:

* Suppose a Lambda function retrieves messages from an SQS queue and processes them.
* If the Lambda function encounters an error during processing, it can configure a visibility timeout for the message to ensure it remains invisible to other consumers while the error is resolved.
* The Lambda function can use long polling to wait for messages, reducing the number of API calls and conserving resources.
* If message processing fails, the Lambda function can retry processing the message after a delay, gradually increasing the delay between retries using exponential backoff.

### Implementation Considerations:

* Configure appropriate visibility timeouts to ensure messages are not processed concurrently by multiple consumers.
* Use long polling to reduce API call costs and improve efficiency, especially in scenarios with low message throughput.
* Implement retry logic with exponential backoff to handle transient errors and ensure message processing resilience.
* Monitor message processing metrics and retry attempts to identify and address any issues in the message processing pipeline.

By leveraging message visibility timeout, long polling, and retries, you can implement a robust and resilient message processing mechanism with Amazon SQS.

**Question 10**

**what is dead letter queue and how long message will stay in aws ?**

ChatGPT

**ChatGPT**

A dead-letter queue (DLQ) is a special type of queue used in message queuing systems, such as Amazon SQS (Simple Queue Service), to handle messages that cannot be processed successfully after a certain number of retries. Here's an overview:

### Dead-Letter Queue (DLQ):

1. **Purpose**:
   * The primary purpose of a dead-letter queue is to capture messages that fail to be processed successfully after a specified number of attempts.
   * Messages may fail processing due to various reasons, including invalid content, processing errors, or application-specific failures.
2. **Retry Limit**:
   * When a message fails processing, the messaging system may automatically retry processing the message for a certain number of times.
   * If the message continues to fail processing after reaching the retry limit, it is considered a "poison" message and is moved to the dead-letter queue.
3. **Error Handling**:
   * Messages in the dead-letter queue can be inspected, analyzed, or manually processed to determine the cause of failure.
   * Dead-letter queues provide a mechanism for handling problematic messages separately from the main processing flow, allowing for easier troubleshooting and resolution.
4. **Configuration**:
   * In AWS services like SQS, you can configure a dead-letter queue for a standard queue by specifying the Amazon Resource Name (ARN) of the dead-letter queue and the maximum number of receive attempts for a message before it's moved to the dead-letter queue.
   * You can configure dead-letter queues for both standard and FIFO queues in SQS.

### Message Retention in AWS SQS:

In Amazon SQS, messages can stay in a queue for a maximum retention period of 14 days. After this period, SQS automatically removes the message from the queue, even if it hasn't been processed. This retention period is configurable at the queue level.

### Summary:

A dead-letter queue provides a mechanism for capturing and handling messages that cannot be processed successfully after a specified number of retries. It allows for separate handling and analysis of problematic messages, aiding in troubleshooting and resolution. In AWS SQS, messages can stay in a queue for up to 14 days by default, after which they are automatically removed.