Explanation

Name: Mamdouh Hazem

ID: 10001816 T|20

1. Overview

This project presents an **event-driven architecture** on AWS to manage and store order notifications using managed services such as **Amazon SNS, SQS, Lambda, DynamoDB**, and **CloudWatch**. It is designed to decouple producers from consumers, provide reliable processing of messages, and ensure fault tolerance through **dead-letter queues** (DLQs). All resources were configured via the AWS Console.

2. System Design

The system follows a flow where:

1. A user (or service) publishes a new order message to **Amazon SNS**.
2. The message is automatically forwarded to **Amazon SQS**, which serves as a buffer queue.
3. When a new message arrives in the SQS queue, it **triggers an AWS Lambda function**.
4. The Lambda function reads and parses the message, then writes it to a **DynamoDB table**.
5. If the Lambda fails to process a message 3 times, the message is rerouted to a **Dead Letter Queue** (DLQ) for later inspection.

This architecture enables asynchronous communication between components and supports failure isolation and debugging via DLQs.

3. Services Used

- **Amazon SNS (Simple Notification Service)**: Used to initiate the workflow by publishing a new order event.

- **Amazon SQS (Simple Queue Service)**: Buffers incoming events for reliable, decoupled processing.
- **Amazon SQS DLQ**: Captures unprocessed or failed messages after multiple retries.
- **AWS Lambda**: Stateless function triggered by SQS to process orders and write them to the database.
- **Amazon DynamoDB**: Stores structured order data in a table called `Orders`.
- **Amazon CloudWatch Logs**: Records execution logs for Lambda functions and helps with debugging and monitoring.

---

4. Implementation Process

All steps were executed using the **AWS Management Console**:

1. **DynamoDB Table**: Created `Orders` with `orderId` as the primary partition key.
2. **SNS Topic**: Created `OrderTopic` to receive new order messages.
3. **SQS Queue**: Created `OrderQueue` to subscribe to the topic.
4. **DLQ**: Created `OrderQueueDLQ` and attached it to `OrderQueue` with a max receive count of 3.
5. **Subscription**: Subscribed `OrderQueue` to `OrderTopic`.
6. **Lambda Function**: Deployed `ProcessOrderLambda` using Python 3.12.

```
import json
import boto3

dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('Orders')

def lambda_handler(event, context):
    for record in event['Records']:
        message = json.loads(record['body'])  # Get message from SQS

        # If message is wrapped by SNS, unwrap it
        if 'Message' in message:
            message = json.loads(message['Message'])
```

```
        print(f"Processing order: {message['orderId']}")

        # Save to DynamoDB
        table.put_item(Item=message)

    return {
        'statusCode': 200,
        'body': json.dumps('Order processed successfully')
    }
```

1. **Permissions**: Lambda granted IAM permissions for SQS, DynamoDB, and CloudWatch.
2. **Code Deployment**: Lambda was coded to parse SQS messages and insert them into DynamoDB.
3. **Trigger**: Configured `OrderQueue` to invoke the Lambda function automatically.
4. **Testing**: A test message was sent to SNS. Successful processing confirmed:

- Message flow from SNS to SQS
- Lambda execution
- Record added in DynamoDB
- Logs available in CloudWatch

---

5. Explanation of Visibility Timeout & DLQ

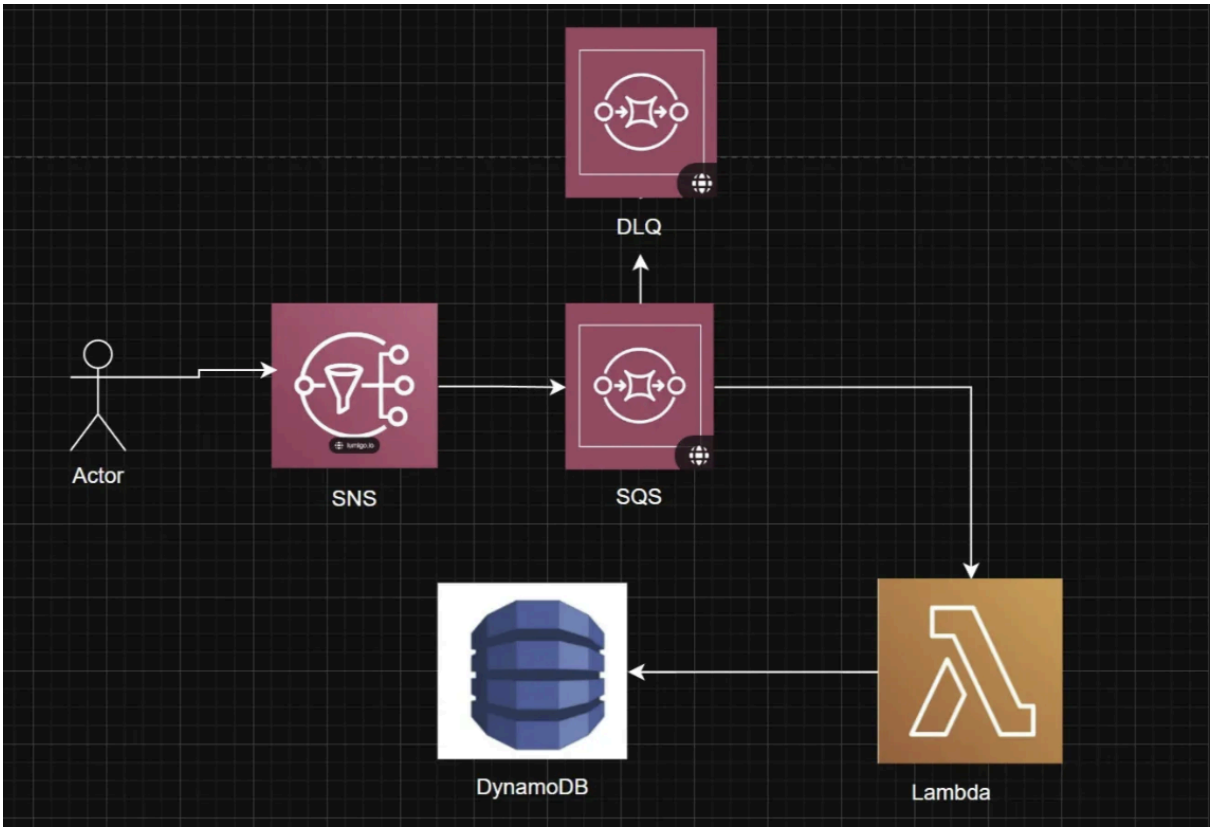
The **visibility timeout** in SQS prevents duplicate processing. When the Lambda function retrieves a message, it becomes hidden from other consumers for a defined time (e.g., 30 seconds). If the function finishes successfully, the message is deleted. If it fails or times out, the message becomes visible again and can be retried.

The **Dead Letter Queue (DLQ)** enhances resilience. If a message fails after the defined max receive count (3), it's moved to the DLQ. This prevents stuck retries and enables manual review of failed events.

Together, visibility timeout and DLQ provide **controlled retry mechanisms**, fault isolation, and **debuggability** without affecting healthy message flows.

6. Architecture Diagram

```
plaintext
CopyEdit
User
 ↓
SNS (OrderTopic)
 ↓
SQS (OrderQueue) → DLQ (OrderQueueDLQ)
 ↓
Lambda (ProcessOrderLambda)
 ↓
DynamoDB (Orders)
```



Each arrow represents a message hand-off between services.

7. CloudFormation Attempt (Optional)

An optional bonus task was attempted using **AWS CloudFormation** to automate deployment. However, the deployment failed due to naming conflicts with existing manually created resources (e.g., `OrderQueueDLQ`, `OrderTopic` already existed). Resolving the issue would require deletion or renaming. Manual setup was retained to validate the system successfully.

---

8. Summary

The solution successfully integrates key AWS services into a **fault-tolerant, event-driven system**. It processes orders in real time, ensures durable storage, and includes error-handling mechanisms. All features and verifications were demonstrated with screenshots.

This project reflects best practices for **scalable**, **decoupled**, and **serverless** architecture using native AWS tools.

Bonus:

.yaml file for cloudformation due to making it using consol:

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Mamdouh Hazem's Event-Driven Order System (Cloud Assignme

Resources:

  OrdersTable:
    Type: AWS::DynamoDB::Table
    Properties:
      TableName: Orders-mhz
      AttributeDefinitions:
        - AttributeName: orderId
          AttributeType: S
      KeySchema:
        - AttributeName: orderId
          KeyType: HASH
      BillingMode: PAY_PER_REQUEST

  OrderTopic:
```

```
  Type: AWS::SNS::Topic
  Properties:
    TopicName: OrderTopic-mhz

OrderQueueDLQ:
  Type: AWS::SQS::Queue
  Properties:
    QueueName: OrderQueueDLQ-mhz

OrderQueue:
  Type: AWS::SQS::Queue
  Properties:
    QueueName: OrderQueue-mhz
    RedrivePolicy:
      deadLetterTargetArn: !GetAtt OrderQueueDLQ.Arn
      maxReceiveCount: 3

OrderQueueSubscription:
  Type: AWS::SNS::Subscription
  Properties:
    TopicArn: !Ref OrderTopic
    Protocol: sqs
    Endpoint: !GetAtt OrderQueue.Arn
    RawMessageDelivery: false

OrderQueuePolicy:
  Type: AWS::SQS::QueuePolicy
  Properties:
    Queues:
      - !Ref OrderQueue
    PolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: Allow
          Principal: "*"
          Action: "SQS:SendMessage"
          Resource: !GetAtt OrderQueue.Arn
          Condition:
```

```
          ArnEquals:
            aws:SourceArn: !Ref OrderTopic

ProcessOrderLambdaRole:
  Type: AWS::IAM::Role
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: Allow
          Principal:
            Service: lambda.amazonaws.com
          Action: sts:AssumeRole
    ManagedPolicyArns:
      - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
      - arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess
      - arn:aws:iam::aws:policy/AmazonSQSFullAccess

ProcessOrderLambda:
  Type: AWS::Lambda::Function
  Properties:
    FunctionName: ProcessOrderLambda-mhz
    Runtime: python3.12
    Handler: index.lambda_handler
    Role: !GetAtt ProcessOrderLambdaRole.Arn
    Code:
      ZipFile: |
        import json
        import boto3

        dynamodb = boto3.resource('dynamodb')
        table = dynamodb.Table('Orders-mhz')

        def lambda_handler(event, context):
            for record in event['Records']:
                message = json.loads(record['body'])
                if 'Message' in message:
                    message = json.loads(message['Message'])
```

```
            print(f"Processing order: {message['orderId']}")
            table.put_item(Item=message)
        return {
            'statusCode': 200,
            'body': json.dumps('Order processed successfully')
        }

  LambdaSQSTrigger:
    Type: AWS::Lambda::EventSourceMapping
    Properties:
      EventSourceArn: !GetAtt OrderQueue.Arn
      FunctionName: !Ref ProcessOrderLambda
      Enabled: true
      BatchSize: 1

Outputs:
  OrdersTableName:
    Description: DynamoDB table name
    Value: !Ref OrdersTable
  OrderTopicArn:
    Description: SNS topic ARN
    Value: !Ref OrderTopic
  OrderQueueURL:
    Description: SQS Queue URL
    Value: !Ref OrderQueue
  OrderQueueDLQURL:
    Description: SQS Dead Letter Queue URL
    Value: !Ref OrderQueueDLQ
  LambdaFunctionName:
    Description: Lambda function name
    Value: !Ref ProcessOrderLambda
```