Event-Driven Order Notification System – Assignment Report

Adham Hisham Salah Hassan

10000480 | T-20

1. Introduction

This project implements an event-driven order notification system on AWS, using several integrated AWS services including **DynamoDB**, **SNS**, **SQS**, **Lambda**, and **CloudWatch**. The system processes incoming order messages asynchronously, reliably stores order data, and handles failures gracefully through a dead letter queue.

All components were manually deployed and configured using the AWS Management Console.

---

2. System Architecture

The system is built around an event-driven architecture with the following workflow:

1. A message representing a new order is **published to an SNS Topic** (`OrderTopic`).
2. SNS forwards the message to an **SQS Queue** (`OrderQueue`), which acts as a buffer.
3. An **AWS Lambda function** (`ProcessOrderLambda`) is triggered automatically when a message arrives in the queue.
4. The Lambda function reads the message and **writes the order data into a DynamoDB table** (`Orders`).
5. If the Lambda function fails to process the message after 3 attempts, the message is redirected to an **SQS Dead Letter Queue** (`OrderQueueDLQ`) for troubleshooting.

This design ensures decoupling between message producers and consumers, supports scalability, and enhances fault tolerance.

---

3. AWS Services Used

The following AWS services were used to implement the system:

- **Amazon DynamoDB**: Stores order records in a table named `Orders`.
- **Amazon SNS**: Publishes notifications when a new order is placed.
- **Amazon SQS**: Queues messages from SNS for asynchronous processing.
- **Amazon SQS Dead Letter Queue**: Stores messages that failed to process after retries.
- **AWS Lambda**: Processes messages from SQS and saves them to DynamoDB.
- **Amazon CloudWatch Logs**: Monitors and logs Lambda execution.

---

4. Implementation Steps

The following steps were completed manually on the AWS Console:

1. Created a **DynamoDB table** named `Orders` with `orderId` as the partition key.
2. Created an **SNS topic** named `OrderTopic`.
3. Created an **SQS queue** named `OrderQueue`.
4. Created a **Dead Letter Queue (DLQ)** named `OrderQueueDLQ`.
5. Configured `OrderQueue` to use `OrderQueueDLQ` as its DLQ with a maximum receive count of 3.
6. Created an **SNS subscription** to send messages from `OrderTopic` to `OrderQueue`.
7. Created a **Lambda function** `ProcessOrderLambda` with runtime Python 3.12.
8. Attached permissions to Lambda allowing access to DynamoDB, SQS, and CloudWatch Logs.
9. Wrote and deployed Lambda code to process SQS messages and write them into DynamoDB.

10. Added an **event source mapping (trigger)** linking `OrderQueue` to `ProcessOrderLambda`.
11. Published a test message to `OrderTopic` and verified:

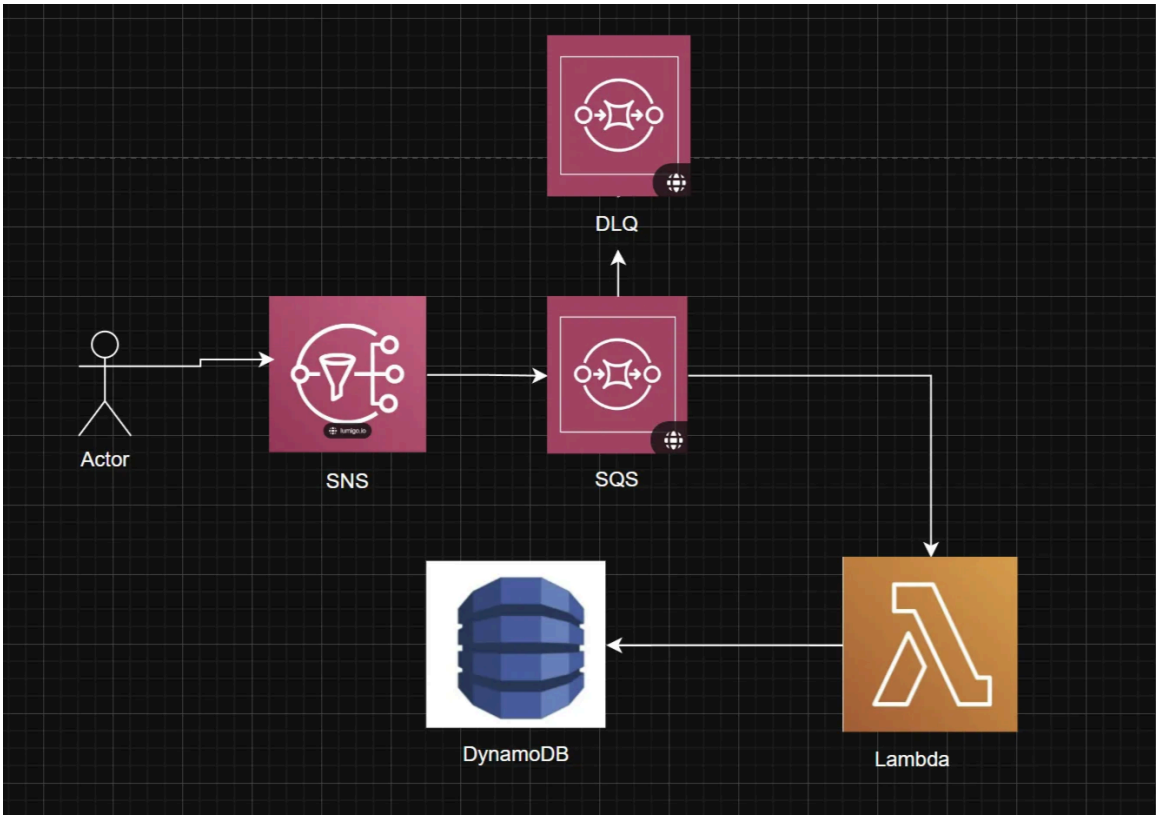
- Message was forwarded to `OrderQueue`.
- Lambda processed the message and inserted data into DynamoDB.
- Logs were recorded in CloudWatch.

All steps were successfully verified, and screenshots are included in the submission file.

---

5. Architecture Diagram

User/Client
↓
SNS (OrderTopic)
↓
SQS (OrderQueue) → DLQ (OrderQueueDLQ)
↓
Lambda (ProcessOrderLambda)
↓
DynamoDB (Orders)



- arrows representing message flow

6. Optional CloudFormation Export

An attempt was made to export the architecture using CloudFormation; however, due to naming conflicts with existing resources (SNS topic and SQS queue names already in use), the stack deployment failed.

Resolving this would require either deleting the existing resources or renaming them in the template.

Since the optional bonus export was not mandatory, the infrastructure was successfully validated through manual deployment instead.

7. Explanation of Design Choices

This architecture follows best practices for building serverless, event-driven applications:

- **Decoupling**: SNS decouples the message producer from downstream consumers.

- **Reliability**: SQS buffers messages to prevent loss during high load or downstream failure.
- **Automatic scaling**: Lambda automatically scales processing based on incoming queue messages.
- **Fault isolation**: A Dead Letter Queue isolates failed messages for manual review.
- **Serverless**: No servers to manage, reducing operational overhead.

Overall, the system provides a scalable, resilient, and maintainable workflow for processing orders asynchronously.

---

8. Conclusion

All required components of the assignment were successfully implemented using the AWS Management Console. The event-driven system was fully tested and verified to process messages end-to-end, storing order data reliably in DynamoDB.

Screenshots of each configuration and verification step are included as supporting documentation.

This solution demonstrates the use of key AWS services for building a decoupled and fault-tolerant serverless architecture.