

Name:Prathamesh Shetty**Class:D15B****Roll No:57**

Case Study Report: Serverless Data Processing with AWS Lambda, S3 and DynamoDB

1. Introduction

This case study demonstrates how to implement a serverless data processing solution using AWS Lambda, S3, and DynamoDB. The solution processes a JSON file uploaded to an S3 bucket, extracts specific fields (such as userID and timestamp), and writes them to a DynamoDB table. This project showcases AWS's serverless capabilities and scalable data processing.

2. Objective

The objective is to create an AWS Lambda function that triggers automatically when a JSON file is uploaded to an S3 bucket. The function parses the file to extract specific fields and writes them to a DynamoDB table. The steps cover the entire process from creating the infrastructure to testing the solution.

3. Tools Used

- **AWS Lambda Function:** A serverless compute service that allows running code without provisioning or managing servers.
 - **Amazon S3:** An object storage service used to store the JSON files.
 - **Amazon DynamoDB:** A fully managed NoSQL database where the extracted fields from the JSON file are stored.
 - **Python:** The programming language used in the Lambda function.
-

4. Step-by-Step Implementation

4.1. Creating the DynamoDB Table

1. Go to the **DynamoDB** service in AWS.
2. Create a new table named **practical**.
3. Enter partition key as **UserID** and sort key as **timestamp**.

Create table

Table details [Info](#)

DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

Table name

This will be used to identify your table.

Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).

Partition key

The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.

1 to 255 characters and case sensitive.

Sort key - optional

You can use a sort key as the second part of a table's primary key. The sort key allows you to sort or search among all items sharing the same partition key.

1 to 255 characters and case sensitive.

4.2. Creating an S3 Bucket

1. Navigate to the **S3** service in AWS.
2. Create a new S3 bucket named **jsonUpload** to upload JSON files.
3. Keep all other settings default.

Create bucket [Info](#)

Buckets are containers for data stored in S3.

General configuration

AWS Region

Asia Pacific (Mumbai) ap-south-1

Bucket name [Info](#)

Bucket name must be unique within the global namespace and follow the bucket naming rules. [See rules for bucket naming](#)

Copy settings from existing bucket - optional

Only the bucket settings in the following configuration are copied.

Format: s3://bucket/prefix

4.3. Writing the AWS Lambda Function in Python

1. Go to **AWS Lambda** and create a new Lambda function.
2. Choose Python 3.x as the runtime.
3. Write the Python code that extracts the **UserID** and **timestamp** from the JSON file.

Create function Info

Choose one of the following options to create your function.

☒ **Author from scratch**
 Start with a simple Hello World example.

☐ **Use a blueprint**
 Build a Lambda application from sample code and configuration presets for common use cases.

☐ **Container image**
 Select a container image to deploy for your function.

Basic information

Function name
Enter a name that describes the purpose of your function.

Function name must be 1 to 64 characters, must be unique to the Region, and can't include spaces. Valid characters are a-z, A-Z, 0-9, hyphens (-), and underscores (_).

Runtime Info
Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.

Architecture Info
Choose the instruction set architecture you want for your function code.

☒ **x86_64**
☐ arm64

Lambda Function code:

```
import json
import boto3

s3 = boto3.client('s3')
dynamodb = boto3.resource('dynamodb')

def lambda_handler(event, context):

    bucket_name = event['Records'][0]['s3']['bucket']['name']
    object_key = event['Records'][0]['s3']['object']['key']

    try:

        s3_response = s3.get_object(Bucket=bucket_name, Key=object_key)
        file_content = s3_response['Body'].read().decode('utf-8')
```

```

json_content = json.loads(file_content)

user_id = json_content.get('UserID')
timestamp = json_content.get('timestamp')

table = dynamodb.Table('practical')
table.put_item(Item={
    'UserID': user_id,
    'timestamp': timestamp
})

return {
    'statusCode': 200,
    'body': 'Data successfully written to DynamoDB!'
}

except Exception as e:
    print(f"Error processing the S3 file: {e}")
    return {
        'statusCode': 500,
        'body': 'Error processing file'
    }

```

4.3. Assigning IAM Roles and Permissions for Lambda

Before setting up the event trigger, it's essential to ensure that your Lambda function has the correct permissions to access both S3 and DynamoDB. Follow these steps to assign the necessary IAM roles and permissions:

- Go to your lambda function and below click on **Configurations** tab.
- Go to **permissions** and click on Role Name link..
- Scroll down and click of **Add Permissions** drop down and select **Attach Policies**.
- Attach the following policies:
 - **AmazonS3ReadOnlyAccess** (to allow Lambda to read objects from S3).
 - **AmazonDynamoDBFullAccess** (to allow Lambda to write to DynamoDB).
- IAM roles are provided to Lambda function to interact with S3 bucket and DynamoDB table.

Permissions policies (3) [Info](#)

You can attach up to 10 managed policies.

Filter by Type

Search All types < 1 > ⚙️

<input type="checkbox"/>	Policy name 🔗	Type	Attached entities
<input type="checkbox"/>	AmazonDynamoDBFullAccess	AWS managed	2
<input type="checkbox"/>	AmazonS3ReadOnlyAccess	AWS managed	2
<input type="checkbox"/>	AWSLambdaBasicExecutionRole-a6cef1b...	Customer managed	1

4.4. Setting Up S3 Event Trigger for Lambda

After setting up the correct permissions, you can now configure the S3 bucket to trigger the Lambda function when a JSON file is uploaded:

- In the **S3 bucket**, navigate to **Properties** and scroll to **Event Notifications**.
- Create a new event notification and select the **Put** event type to trigger the function when an object is created (uploaded).
- Specify the **prefix** (if applicable) to limit the event to only certain file types (e.g., `*.json`).
- In the **Lambda Trigger** section, choose the Lambda function you previously created, and save the configuration.

Create event notification [Info](#)

To enable notifications, you must first add a notification configuration that identifies the events you want Amazon S3 to publish and the destinations where you want Amazon S3 to send the notifications.

General configuration

Event name

 Event name can contain up to 255 characters.

Prefix - optional
 Limit the notifications to objects with key starting with specified characters.

Suffix - optional
 Limit the notifications to objects with key ending with specified characters.

Event types

Specify at least one event for which you want to receive notifications. For each group, you can choose an event type for all events, or you can choose one or more individual events.

Object creation

☒ All object create events ☐ Put

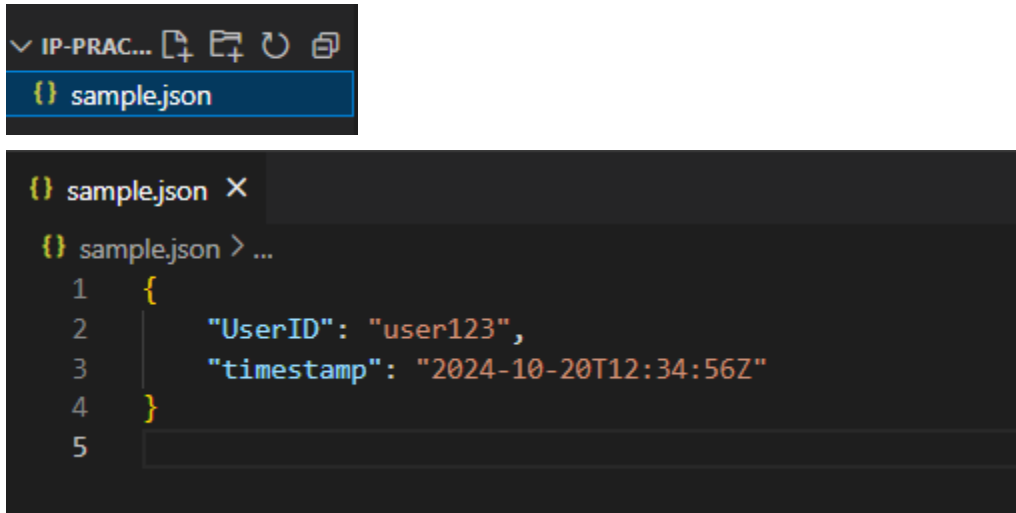
Event notifications (1) [Learn more](#) [🔗](#)

Send a notification when specific events occur in your bucket.

<input type="checkbox"/>	Name	Event types	Filters	Destination type	Destination
<input type="checkbox"/>	jsonUpload	All object create events	.json	Lambda function	pythonfunction 🔗

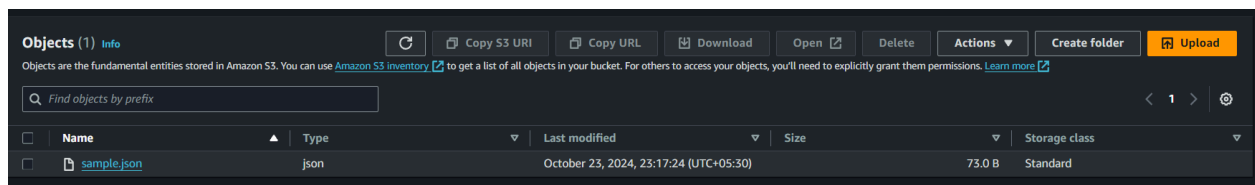
4.5. Uploading a Sample JSON File to S3

1. Create a sample JSON file with UserID and timestamp values.



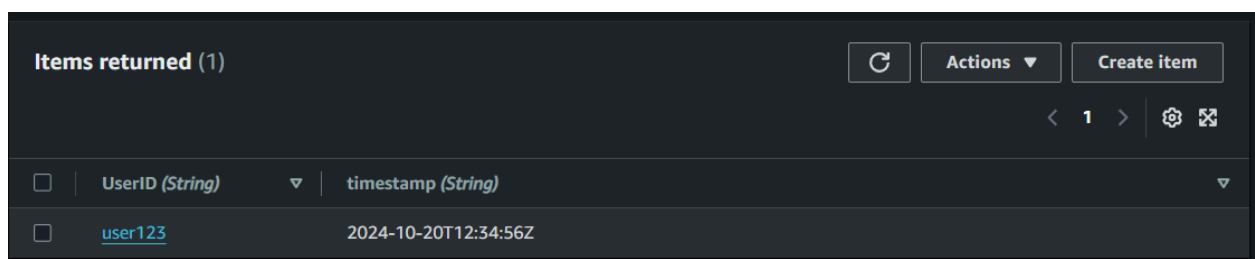
```
{
  "UserID": "user123",
  "timestamp": "2024-10-20T12:34:56Z"
}
```

2. Upload this file to the **jsonUpload** S3 bucket.



4.6. Verifying Data in DynamoDB

1. Go to the **DynamoDB** console.
2. View the data in the **practical** table and verify that the UserID and timestamp were written successfully.



5. Key Feature and Application

The unique key feature of this case study is the **serverless, event-driven architecture** provided by AWS Lambda. This enables automatic execution of functions in response to events, such as an S3 file upload, without the need to manage or provision servers.

Application in IoT Data Collection and Processing:

In current time IoT devices use Traditional Server-Based Architecture. IoT devices continuously send data to centralized servers, which can be hosted either on-premise or in the cloud. These servers listen for incoming data, store it, and perform analysis. However, managing this infrastructure presents significant challenges, including provisioning resources, scaling servers to handle varying data loads, and ensuring high availability.

This process can be made very convenient using Lambda Function. In IoT applications, devices generate a continuous stream of data. By using a similar event-driven architecture, data from sensors or devices can be uploaded to S3 and automatically processed by Lambda. This system can analyze or store data in DynamoDB for further use, such as generating alerts, dashboards, or usage reports.

6. Conclusion

This case study illustrates the efficiency and scalability of AWS's serverless infrastructure in handling real-time data processing with minimal operational complexity. By utilizing S3 for storage, Lambda for event-triggered processing, and DynamoDB for persistent storage, we create a seamless, highly reliable solution. The architecture scales automatically to handle varying workloads, reducing the need for manual infrastructure management. This setup is particularly powerful for IoT applications, where devices generate continuous streams of data, as it allows for automatic data collection, analysis, and storage without the overhead of maintaining servers or complex scaling solutions.