Case Study

Leveraging AWS Lambda, S3, and DynamoDB for Serverless Data Processing

Name: Prajyot Shinde

Roll No: 57 (D15A)

Introduction:

In this case study, we explore the benefits, challenges, and practical applications of building a serverless architecture using **AWS Lambda**, **Amazon S3**, and **Amazon DynamoDB**. These services are a part of the AWS ecosystem and allow developers to create scalable, low-maintenance systems that automatically adjust to varying loads and are event-driven by design.

The primary use case for this study is to process files (JSON) uploaded to **S3**, trigger **AWS Lambda** to extract relevant data from the JSON files, and store the data in a **DynamoDB** table. This approach will be compared with traditional methods that use databases such as **MongoDB** and **PostgreSQL**, illustrating the pros, cons, and alternative solutions.

Background:

Traditionally, applications rely on servers running continuously to handle requests, perform business logic, and manage storage through databases. This can become costly and complex as the infrastructure scales. Moreover, applications that need to process data on demand often require careful capacity planning and load management.

The evolution of cloud computing and serverless technologies has introduced new ways of handling backend workloads. **AWS Lambda** removes the need to manage servers, offering on-demand code execution. Combined with **Amazon S3** for file storage and **DynamoDB** for NoSQL database services, developers can create serverless, event-driven applications that dynamically scale with user demands.

In this case study, we examine an example scenario to demonstrate how Lambda, S3, and DynamoDB are integrated and why this combination is ideal for certain applications.

Scenario Overview

The scenario involves developing an event-driven system where users upload JSON files containing user information (such as userID, timestamp, etc.) to an S3 bucket. Upon the upload, an AWS Lambda function is triggered to:

1. **Read the file** from S3.

2. **Parse the JSON data** to extract specific fields like userID and timestamp.
3. **Store this data** in a DynamoDB table.

We will look at the steps involved, the benefits of using this architecture, and the limitations compared to traditional systems.

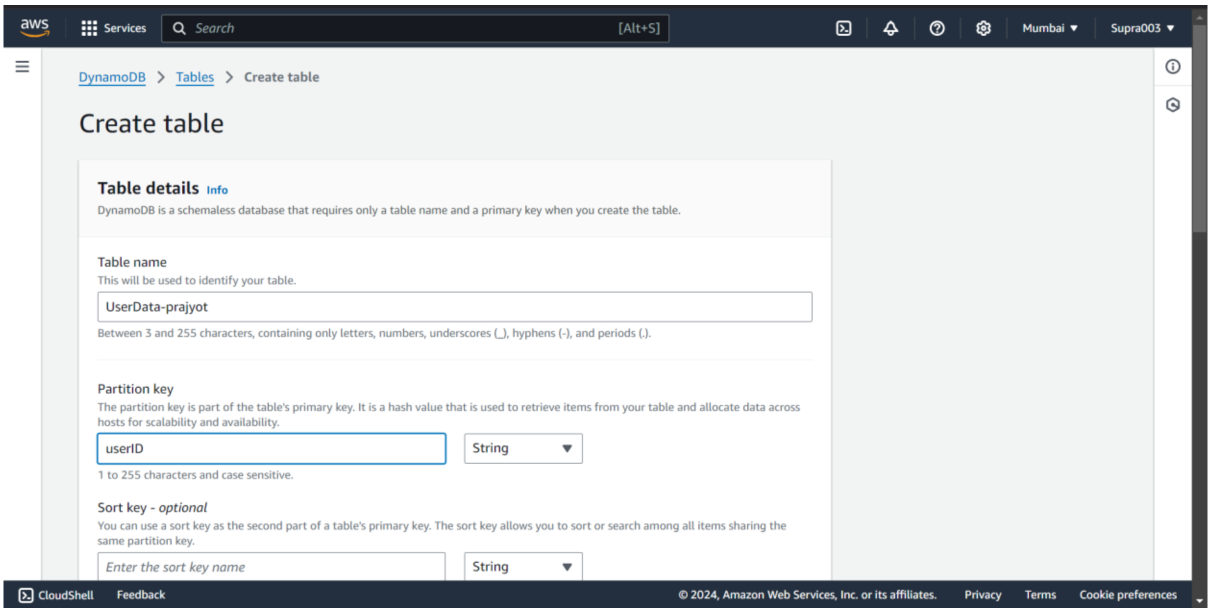
Architecture Diagram:

```
[Client Uploads JSON]      ->      [S3 Bucket]   ->      [Lambda Function]
                                                 ->      [DynamoDB Table]
```

Step-by-Step Breakdown:

Step 1: Create a DynamoDB Table

1. **Log in to the AWS Management Console**:
   - Go to the AWS Management Console.
2. **Navigate to DynamoDB**:
   - In the services menu, search for **DynamoDB** and click on it.
3. **Create a new table**:
   - Click on **Create table**.
   - **Table name**: Choose a name for your table, e.g., UserData.
   - **Partition key**: Define a partition key, e.g., userID (String type).
   - Optionally, you can define a sort key (e.g., timestamp), but for simplicity, we will only use a partition key here.
   - Leave the rest of the settings as default, but feel free to customize them based on your needs.
   - Click on **Create**.


DynamoDB > Tables > Create table

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.

UserData-prajyot

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.

userID | String

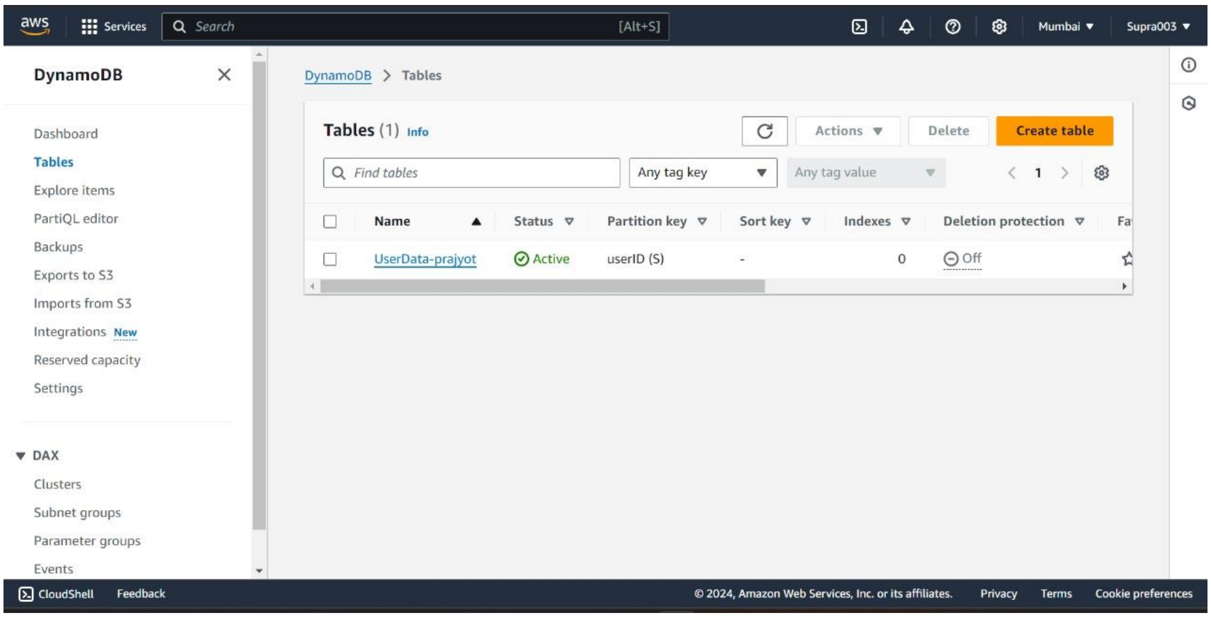
1 to 255 characters and case sensitive.

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.

Enter the sort key name | String

4. **Define Attributes**:
   - Add relevant attributes based on your JSON structure. For this case, you may need attributes like userID, timestamp, and any other fields you plan to store.


DynamoDB > Tables

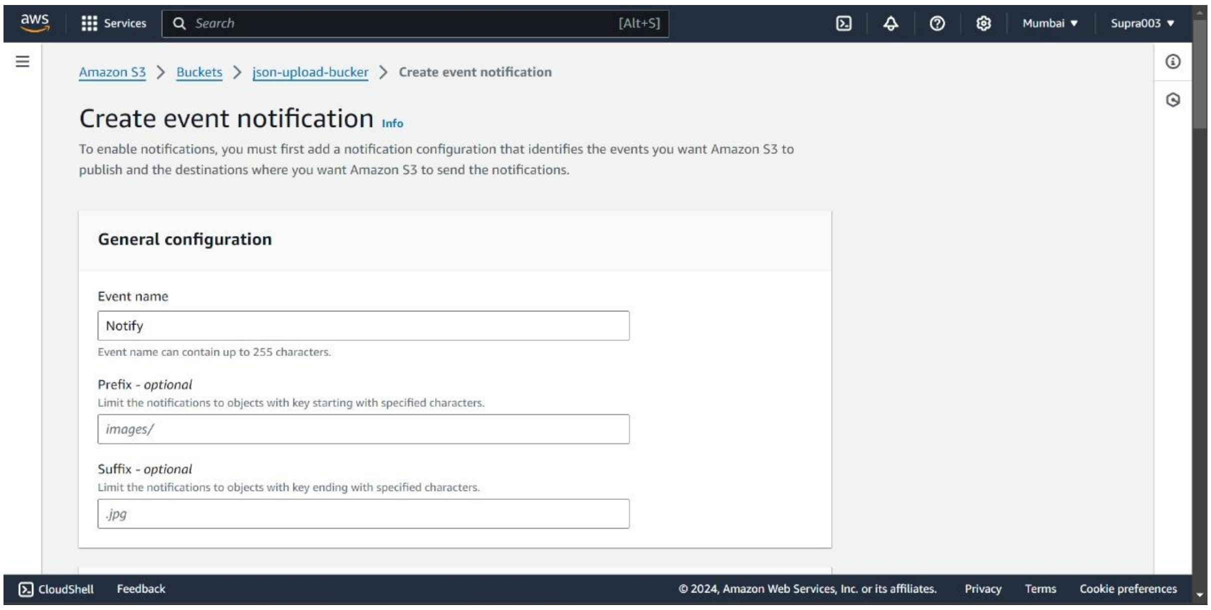
Tables (1) Info

Name	Status	Partition key	Sort key	Indexes	Deletion protection
UserData-prajyot	Active	userID (S)	-	0	Off

Step 2: Create an S3 Bucket

1. **Navigate to S3**:
   - In the AWS Management Console, search for **S3** and click on it.
2. **Create a new bucket**:
   - Click on **Create bucket**.
   - **Bucket name**: Choose a unique name (e.g., my-json-bucket).

- **Region**: Select a region where you want to create the bucket.
- Leave other options as default or customize as necessary.
- Click **Create bucket**.



Amazon S3 > Buckets > json-upload-bucker > Create event notification

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.

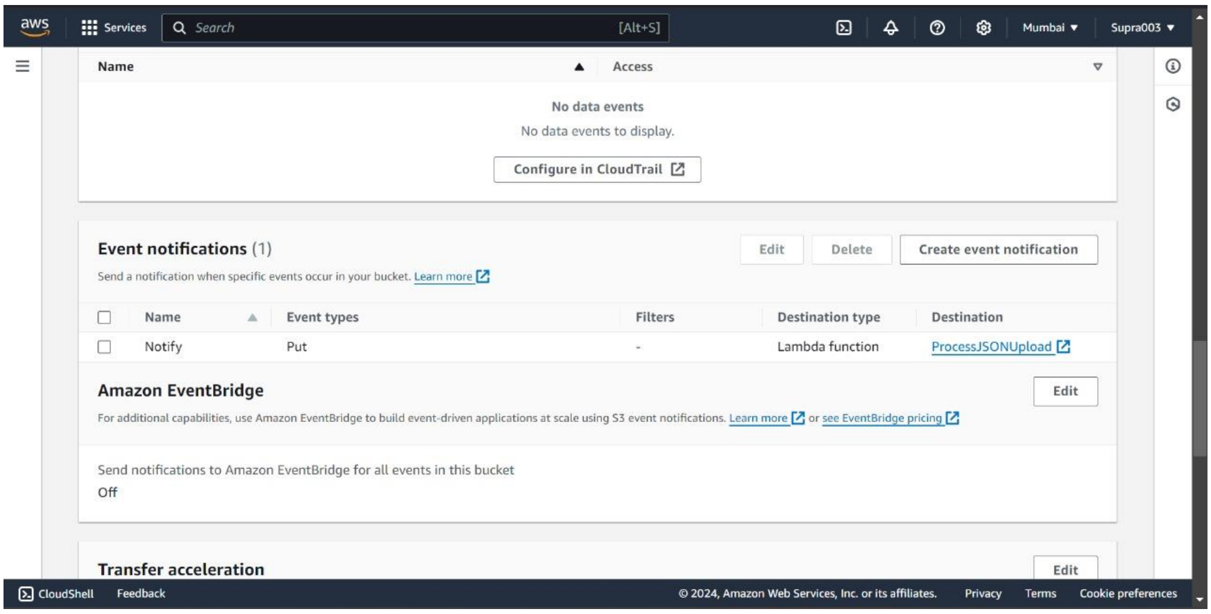
General configuration

Event name: Notify
Event name can contain up to 255 characters.

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

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

© 2024, Amazon Web Services, Inc. or its affiliates.



No data events
No data events to display.

Event notifications (1)
Send a notification when specific events occur in your bucket. Learn more

Name	Event types	Filters	Destination type	Destination
Notify	Put	-	Lambda function	ProcessJSONUpload

Amazon EventBridge
For additional capabilities, use Amazon EventBridge to build event-driven applications at scale using S3 event notifications. Learn more or see EventBridge pricing

Send notifications to Amazon EventBridge for all events in this bucket
Off

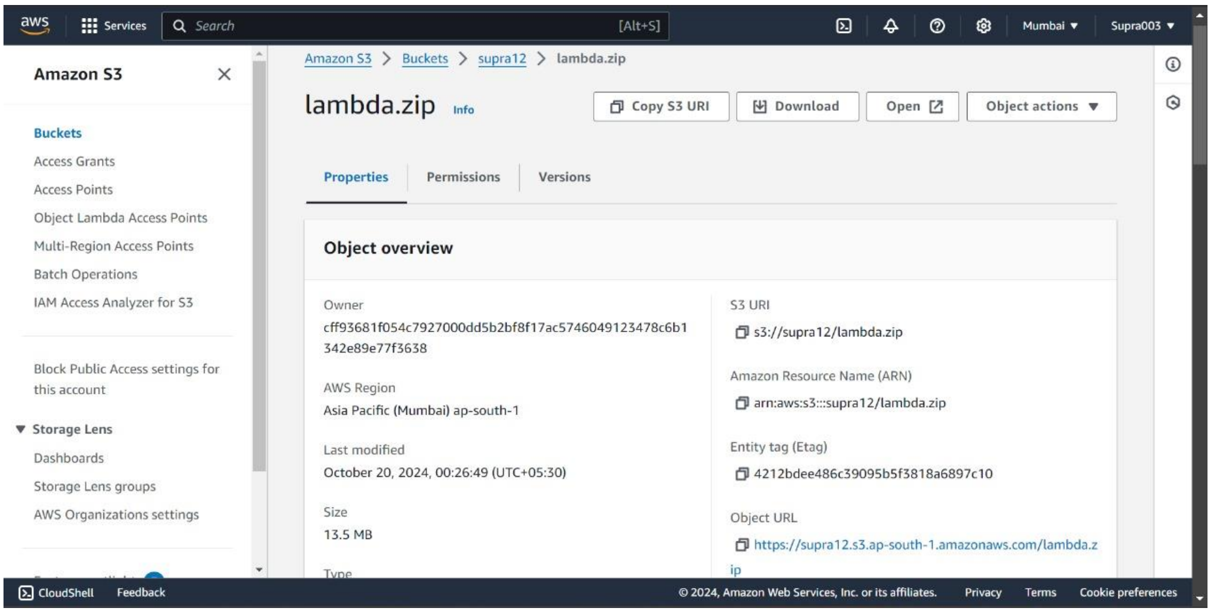
Transfer acceleration

Steps to Upload Your Lambda Function to S3

1. **Upload the ZIP File to S3**:
   - **Open the Amazon S3 Console**: Go to the Amazon S3 console.
   - **Create a New Bucket (if needed)**: If you don't already have a bucket, you can create one:

        - Click on **Create bucket**.
        - Provide a unique bucket name and configure settings as needed.
    - **Upload the ZIP File**:
        - Click on the bucket you want to use.
        - Click on **Upload**.
        - Click on **Add files** and select your ZIP file.
        - Click on **Upload**.

2. **Get the S3 Object URL**: After uploading, find the object in your S3 bucket, and copy its URL. It should look something like this:



Object overview

Owner
cff93681f054c7927000dd5b2bf8f17ac5746049123478c6b1342e89e77f3638

AWS Region
Asia Pacific (Mumbai) ap-south-1

Last modified
October 20, 2024, 00:26:49 (UTC+05:30)

Size
13.5 MB

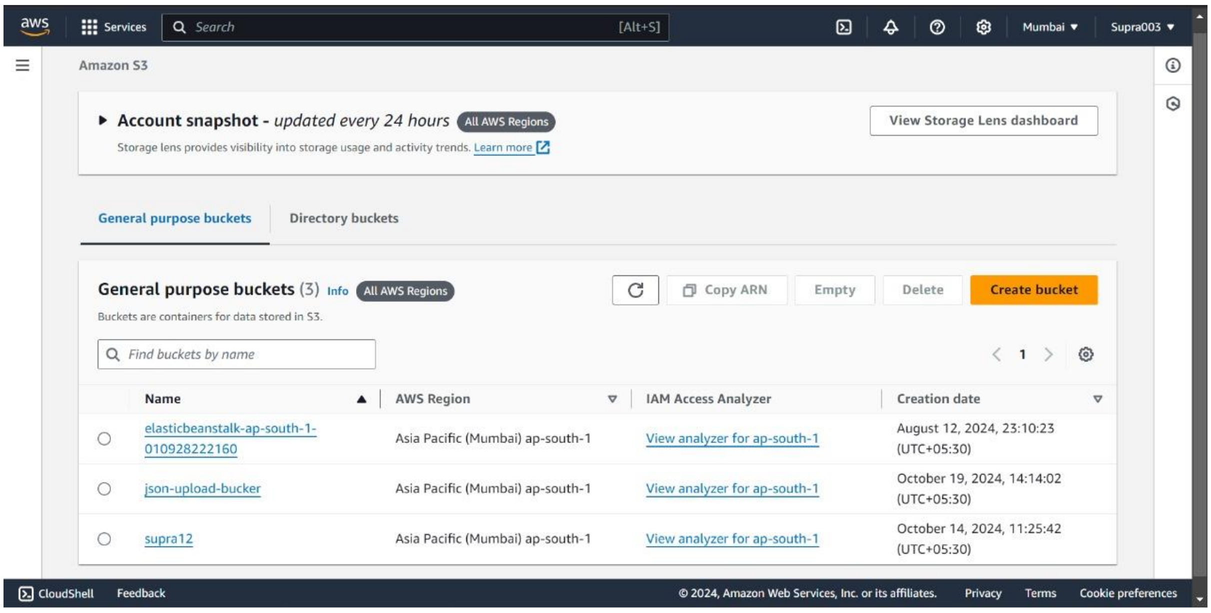
Type

S3 URI
s3://supra12/lambda.zip

Amazon Resource Name (ARN)
arn:aws:s3:::supra12/lambda.zip

Entity tag (Etag)
4212bdee486c39095b5f3818a6897c10

Object URL
https://supra12.s3.ap-south-1.amazonaws.com/lambda.zip



Account snapshot - *updated every 24 hours* All AWS Regions

Storage lens provides visibility into storage usage and activity trends. Learn more

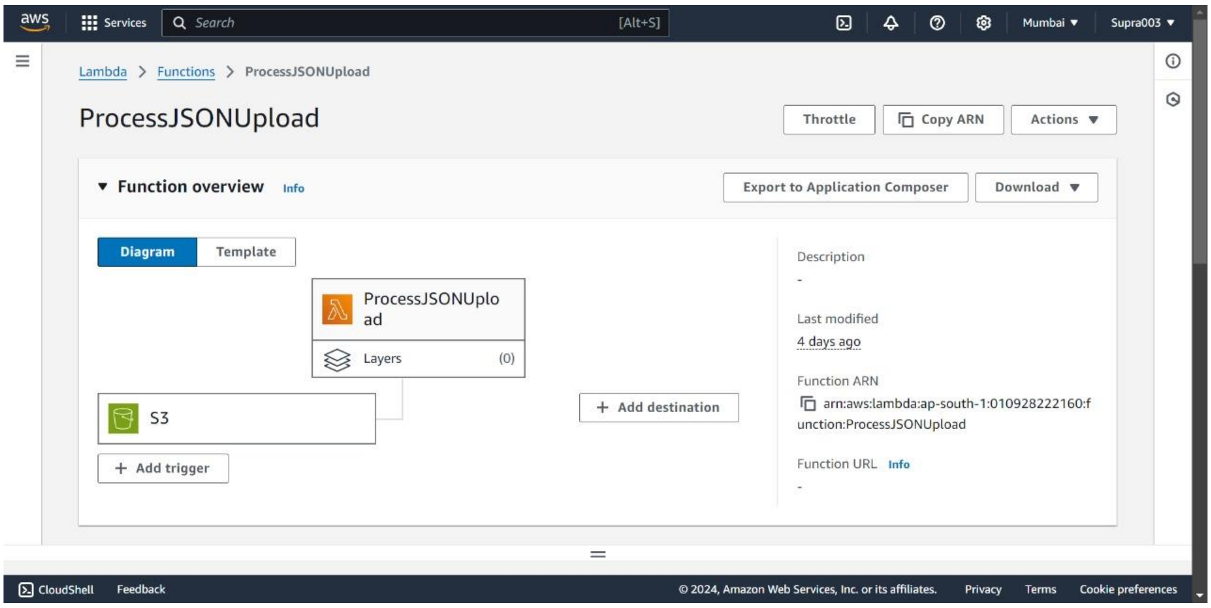
General purpose buckets (3) Info All AWS Regions

Buckets are containers for data stored in S3.

Name	AWS Region	IAM Access Analyzer	Creation date
elasticbeanstalk-ap-south-1-010928222160	Asia Pacific (Mumbai) ap-south-1	View analyzer for ap-south-1	August 12, 2024, 23:10:23 (UTC+05:30)
json-upload-bucker	Asia Pacific (Mumbai) ap-south-1	View analyzer for ap-south-1	October 19, 2024, 14:14:02 (UTC+05:30)
supra12	Asia Pacific (Mumbai) ap-south-1	View analyzer for ap-south-1	October 14, 2024, 11:25:42 (UTC+05:30)

Step 3: Write the Lambda Function

1. **Navigate to Lambda**:
   - In the AWS Management Console, search for **Lambda** and click on it.
2. **Create a new Lambda function**:
   - Click on **Create function**.
   - **Function name**: Give it a name (e.g., ProcessJsonFunction).
   - **Runtime**: Select **Node.js 18.x** (or the latest version).
   - **Permissions**: Choose "Create a new role with basic Lambda permissions" or use an existing role if you have one.



Lambda > Functions > ProcessJSONUpload

ProcessJSONUpload

ProcessJSONUpload

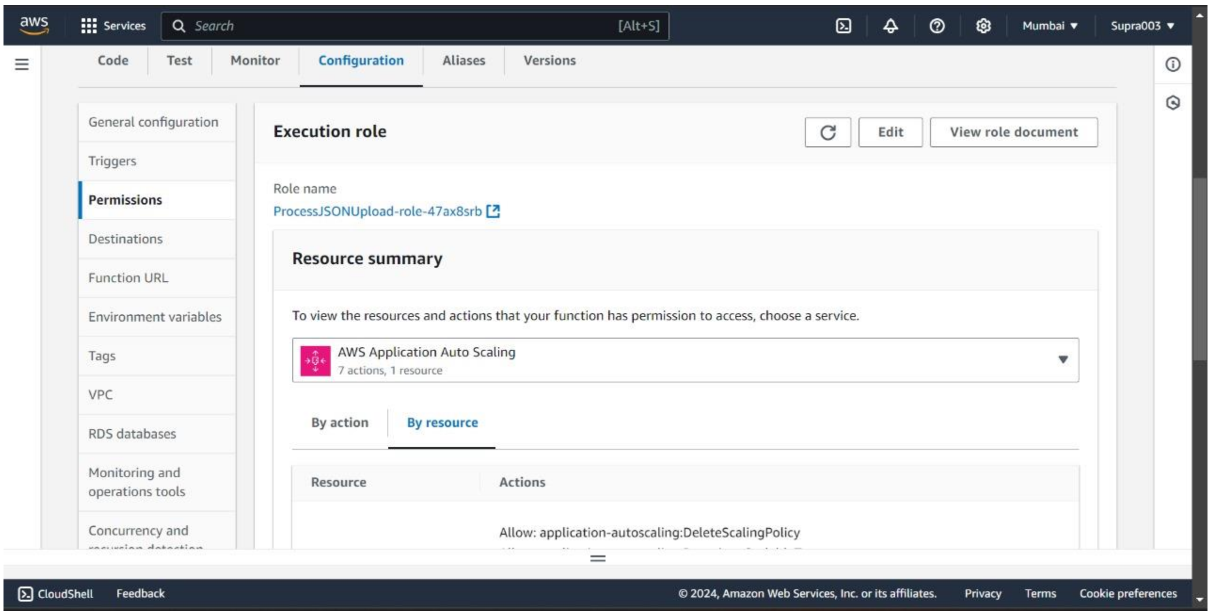
Description
-

Last modified
4 days ago

Function ARN
arn:aws:lambda:ap-south-1:010928222160:function:ProcessJSONUpload

Function URL Info
-

3. **Add the necessary permissions**:
   - Go to the **Permissions** tab of your Lambda function.
   - Click on the role linked to your Lambda function to open the IAM role settings.
   - Click on **Add permissions** > **Attach policies**.
   - Search for and attach the following policies:
     - **AmazonDynamoDBFullAccess** (or create a custom policy with limited access to your table).
     - **AmazonS3ReadOnlyAccess** (or create a custom policy to read from your specific bucket).



Code | Test | Monitor | Configuration | Aliases | Versions

General configuration
Triggers
Permissions
Destinations
Function URL
Environment variables
Tags
VPC
RDS databases
Monitoring and operations tools
Concurrency and recursion detection

Execution role

Role name
ProcessJSONUpload-role-47ax8srb

Resource summary

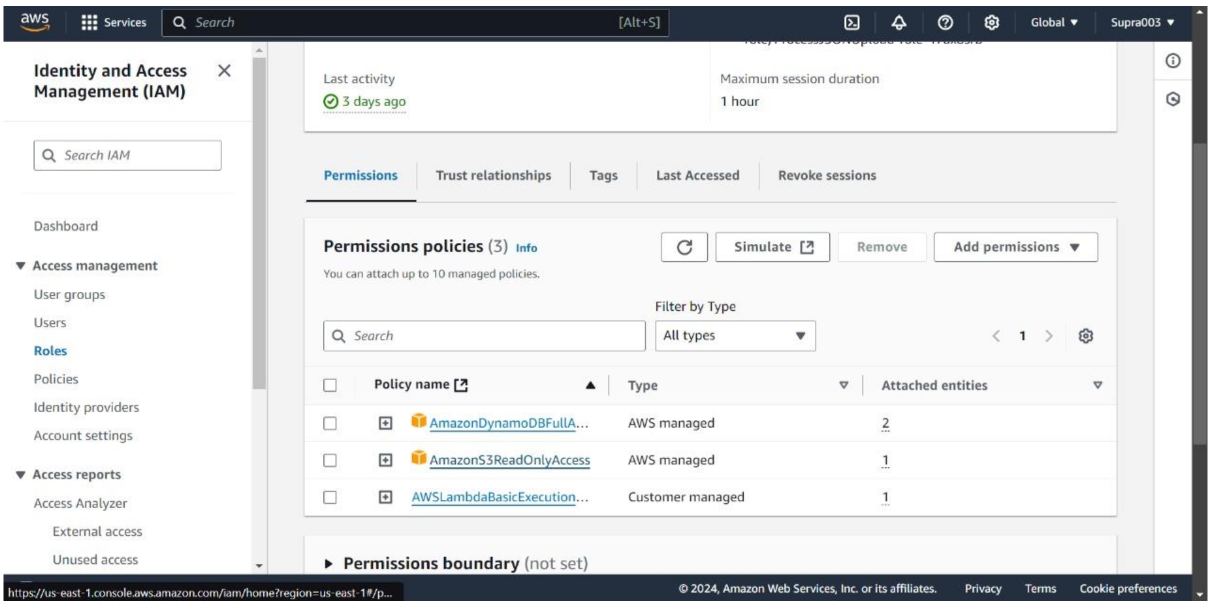
To view the resources and actions that your function has permission to access, choose a service.

AWS Application Auto Scaling
7 actions, 1 resource

By action | By resource

Resource	Actions
	Allow: application-autoscaling:DeleteScalingPolicy

© 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences



Identity and Access Management (IAM)

Dashboard
Access management
User groups
Users
Roles
Policies
Identity providers
Account settings
Access reports
Access Analyzer
External access
Unused access

Last activity: 3 days ago
Maximum session duration: 1 hour

Permissions | Trust relationships | Tags | Last Accessed | Revoke sessions

Permissions policies (3) Info

You can attach up to 10 managed policies.

Filter by Type: All types

Policy name	Type	Attached entities
AmazonDynamoDBFullA...	AWS managed	2
AmazonS3ReadOnlyAccess	AWS managed	1
AWSLambdaBasicExecution...	Customer managed	1

Permissions boundary (not set)

https://us-east-1.console.aws.amazon.com/iam/home?region=us-east-1#/p...

© 2024, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

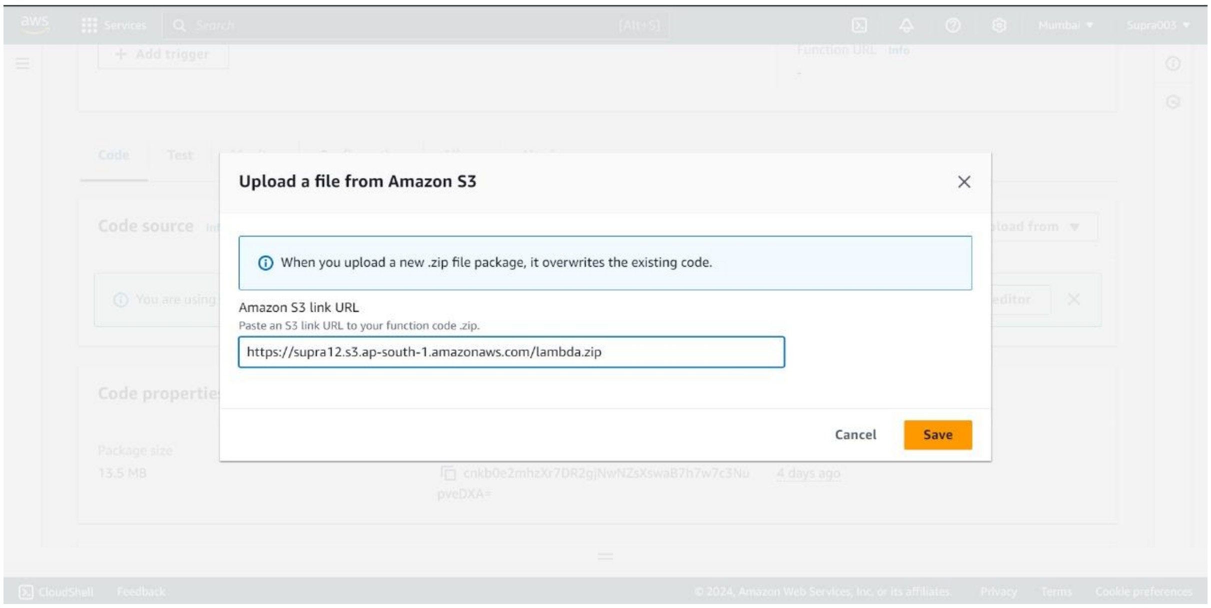
Go to the Function Code Section:

- In the **Code** section, look for the option that allows you to upload from S3.

Enter the S3 URL:

- Paste the S3 object URL you copied earlier into the field for the S3 location.

Save or Update the Function: After entering the S3 URL, click on **Save** or **Deploy**.

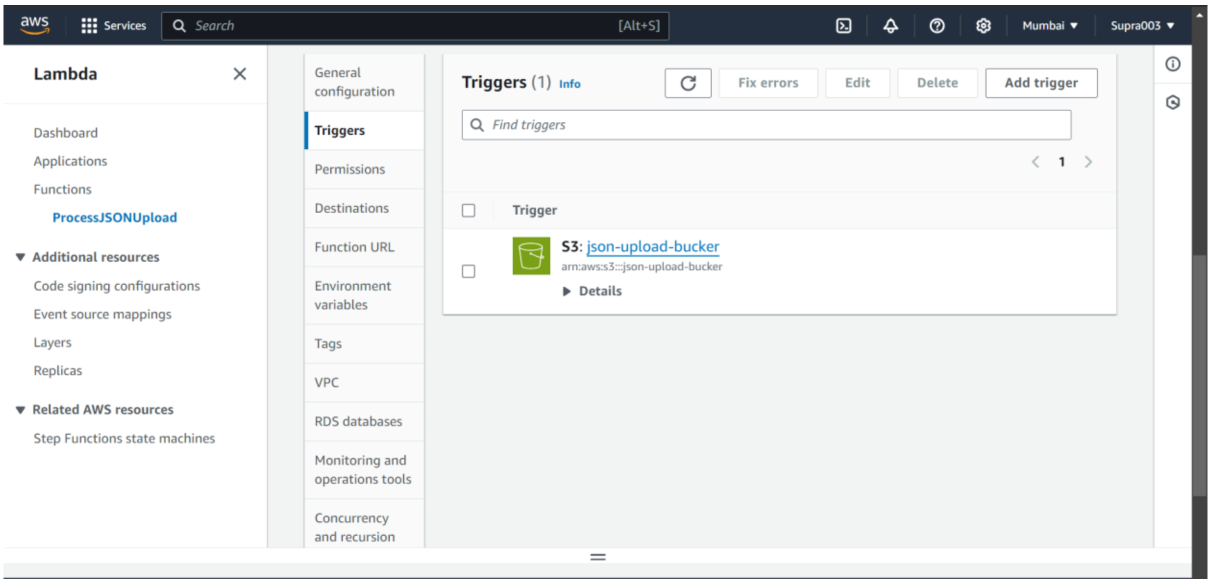

Upload a file from Amazon S3

When you upload a new .zip file package, it overwrites the existing code.

Amazon S3 link URL
Paste an S3 link URL to your function code .zip.
https://supra12.s3.ap-south-1.amazonaws.com/lambda.zip

Step 4: Configure the S3 Trigger

1. **Set up the trigger**:
   - Go to the **Configuration** tab of your Lambda function.
   - Click on **Triggers** and then **Add trigger**.
   - Select **S3** from the dropdown.
   - Choose the bucket you created (my-json-bucket).
   - For the event type, select **All object create events**.
   - Click on **Add**


Triggers (1)

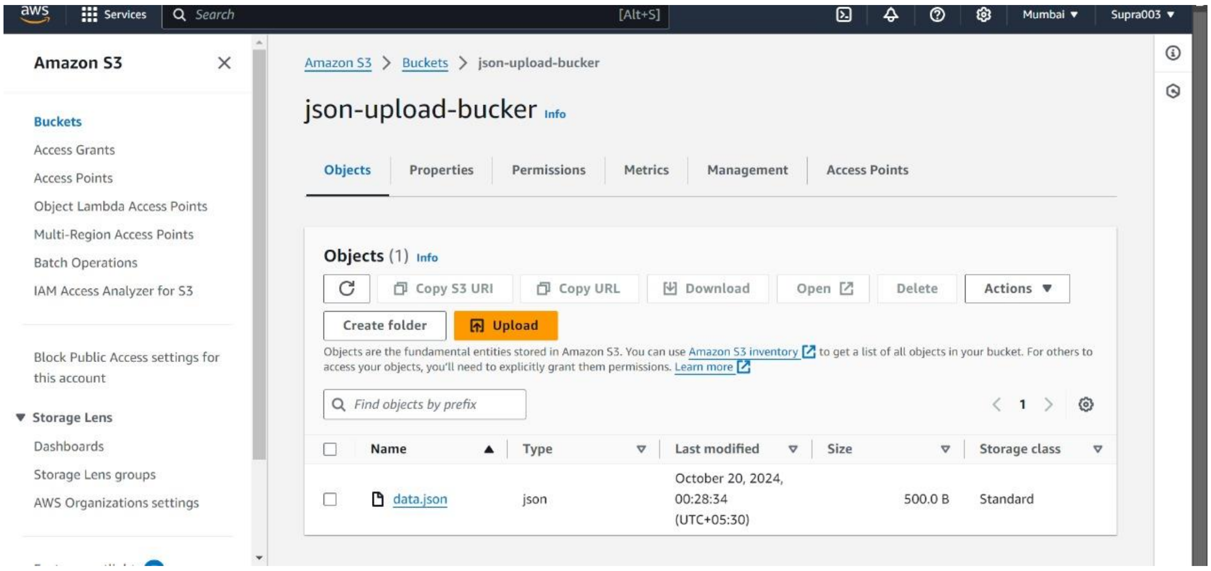
S3: json-upload-bucker
arn:aws:s3:::json-upload-bucker

Step 5: Upload a Sample JSON File to S3

1. **Navigate to your S3 bucket**:
   - Go back to your S3 bucket (my-json-bucket).
2. **Upload a sample JSON file**:
   - Click on **Upload**.
   - Choose **Add files**, and select a sample JSON file with the following structure:

```
[
  {
    "userID": "user123",
    "timestamp": "2024-10-18T10:30:00Z",
    "name": "Prajyot Shinde",
    "email": "prajyot.shinde@example.com",
    "age": 22
      },
      {
        "userID": "user456",
        "timestamp": "2024-10-18T11:00:00Z",
        "name": "Jane Smith",
        "email": "jane.smith@example.com",
        "age": 25
      },
      {
        "userID": "user789",
        "timestamp": "2024-10-18T11:30:00Z",
        "name": "Mike Johnson",
        "email": "mike.johnson@example.com",
        "age": 40
      }
    ]

```



json-upload-bucker

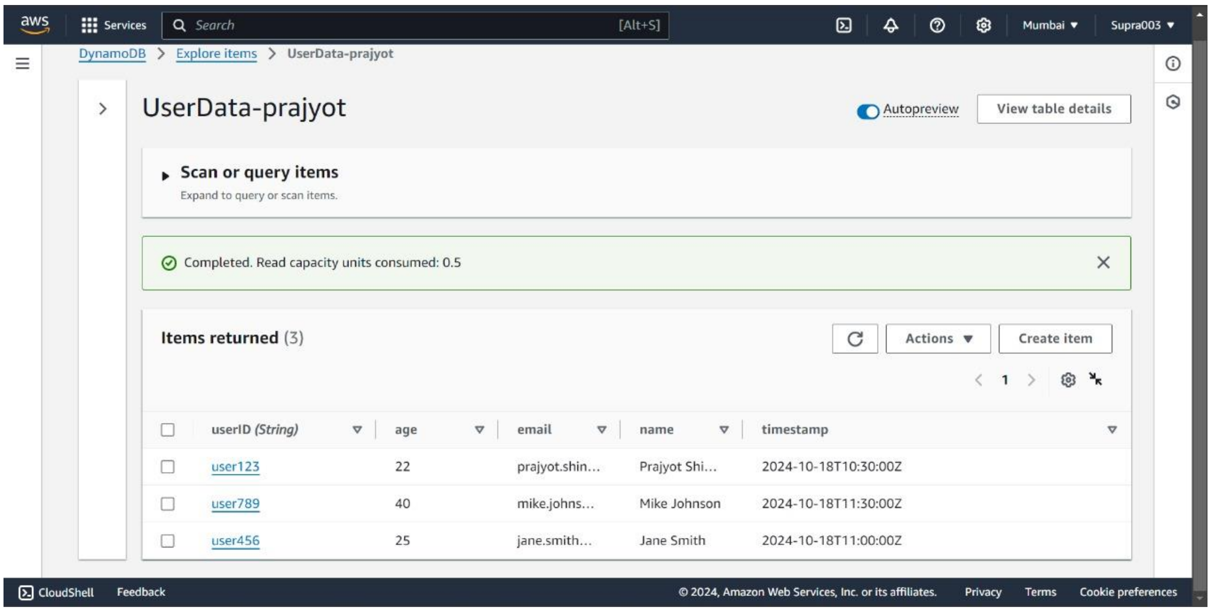
Objects (1)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
data.json	json	October 20, 2024, 00:28:34 (UTC+05:30)	500.0 B	Standard

Step 6: Verify Data in DynamoDB

1. **Navigate to DynamoDB**:
   - In the AWS Management Console, go back to DynamoDB.
2. **View the items in your table**:
   - Click on the table you created (UserData-prajyot).
   - Go to the **Items** tab to see if the data from the JSON file has been successfully written to your DynamoDB table.

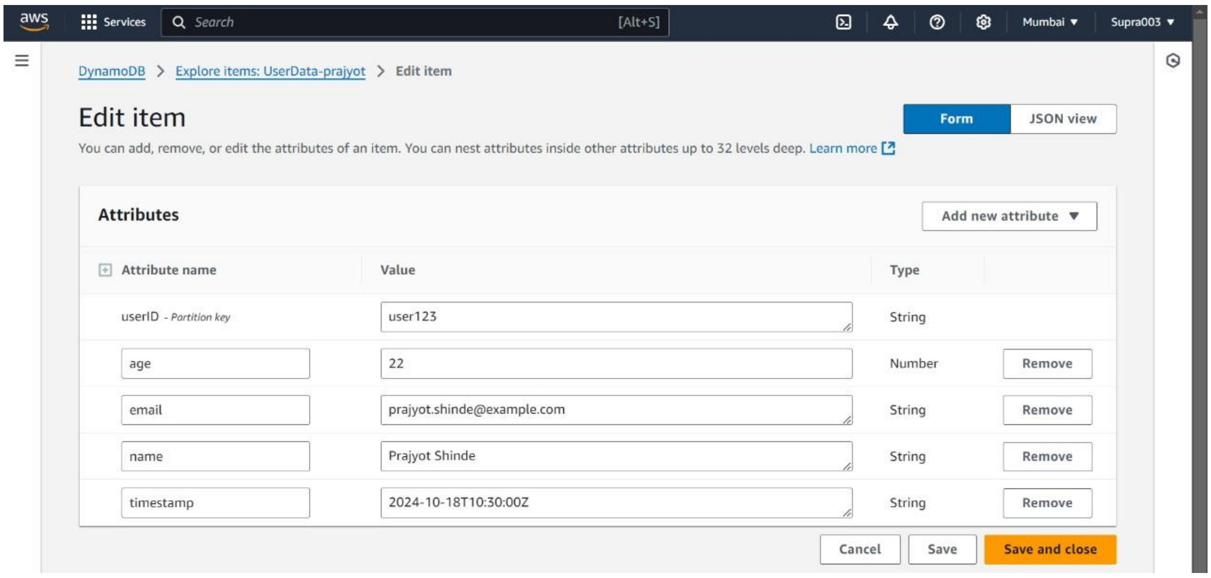

DynamoDB > Explore items > UserData-prajyot
Autopreview
View table details
Scan or query items
Expand to query or scan items.
Actions
Create item
CloudShell
Feedback
© 2024, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

UserData-prajyot

Completed. Read capacity units consumed: 0.5

Items returned (3)

userID (String)	age	email	name	timestamp
user123	22	prajyot.shin…	Prajyot Shi…	2024-10-18T10:30:00Z
user789	40	mike.johns…	Mike Johnson	2024-10-18T11:30:00Z
user456	25	jane.smith…	Jane Smith	2024-10-18T11:00:00Z



Edit item

You can add, remove, or edit the attributes of an item. You can nest attributes inside other attributes up to 32 levels deep. Learn more

Attributes

Attribute name	Value	Type	
userID - Partition key	user123	String	
age	22	Number	Remove
email	prajyot.shinde@example.com	String	Remove
name	Prajyot Shinde	String	Remove
timestamp	2024-10-18T10:30:00Z	String	Remove

Benefits of AWS Lambda, S3, and DynamoDB Architecture

AWS Lambda:

- **Serverless**: There is no need to manage or provision any infrastructure. AWS handles all the scaling, maintenance, and updates.
- **Event-Driven**: Lambda functions can automatically trigger based on predefined events, such as an S3 file upload or a DynamoDB stream.
- **Cost-Efficient**: You only pay for the compute time when the Lambda function is executed, making it highly cost-effective for applications with sporadic usage.

Amazon S3:

- **Scalability**: S3 scales effortlessly as the amount of data increases, without any intervention required from the user.
- **Durability**: S3 offers 99.999999999% durability (11 nines), ensuring that data is protected from data loss.
- **Integration with Lambda**: S3 natively integrates with Lambda, which allows files to trigger backend processing with ease.

Amazon DynamoDB:

- **Low Latency**: DynamoDB provides single-digit millisecond response times, which makes it suitable for applications requiring fast access to data.
- **Scalability**: Like S3, DynamoDB automatically scales with the load, so it can handle a high volume of data without manual intervention.
- **Flexible NoSQL**: DynamoDB is schema-less, which allows you to store and query JSON-like documents flexibly.

Limitations of AWS Lambda, S3, and DynamoDB

AWS Lambda:

- **Cold Starts**: When a Lambda function has not been invoked for some time, there may be a delay (cold start) when it's triggered again.
- **Execution Limits**: Lambda has a maximum execution time of 15 minutes, so long-running processes are not ideal.
- **Stateless**: Lambda functions are stateless, meaning you must use external services like S3 or DynamoDB to store state between executions.

Amazon S3:

- **Eventual Consistency**: S3 follows an eventual consistency model for some operations, such as overwriting an object or listing buckets.
- **Not a File System**: While S3 excels in object storage, it's not suited for high-frequency, small data transactions like a traditional file system.

Amazon DynamoDB:

- **Complex Queries**: DynamoDB is optimized for simple key-value lookups and basic queries. Complex queries with joins or aggregates are not supported natively.
- **Pricing Model**: Costs can increase for applications with high write and read throughput if not carefully optimized.

Use Cases of AWS Lambda, S3, and DynamoDB

- **Serverless Web Applications**: Frontend web applications can store files in S3, process them with Lambda, and store metadata in DynamoDB, creating a completely serverless architecture.
- **IoT Systems**: IoT devices can send data to S3 or DynamoDB through Lambda, providing scalable data processing pipelines.
- **Real-time Analytics**: Event-driven systems like Lambda can be used to process and analyze data in real-time, triggered by uploads or changes in data streams.

Alternatives to AWS Lambda, S3, and DynamoDB

Serverless Compute Alternatives:

- **Google Cloud Functions**: Google Cloud’s serverless offering similar to AWS Lambda.
- **Azure Functions**: Microsoft’s serverless compute option with seamless integration into the Azure ecosystem.

Object Storage Alternatives:

- **Google Cloud Storage**: Similar to Amazon S3 but on Google Cloud, with comparable scalability and durability.
- **Azure Blob Storage**: Microsoft’s cloud object storage service.

NoSQL Alternatives:

- **MongoDB Atlas**: A managed version of MongoDB offering rich query capabilities.
- **Cassandra**: An open-source NoSQL database designed for distributed data storage.

Problems Faced and Solutions

1. **Error: "Cannot find module 'index'"**
    - **Problem**: This error occurred due to AWS Lambda not recognizing the index.mjs file when attempting to load the handler function.
    - **Solution**: The issue was resolved by ensuring that the **handler function** was correctly set in the AWS Lambda configuration and the **filename** was properly named index.mjs in both the ZIP file and AWS Lambda's settings.
2. **Packaging Node Modules**
    - **Problem**: I initially faced issues related to missing the node_modules directory and other dependencies when deploying the Lambda function.
    - **Solution**: By ensuring the **aws-sdk** was installed and packaging only the necessary files (index.mjs and package.json) into the ZIP file, I overcame this issue. The **aws-sdk** is pre-installed in AWS Lambda, so no need to include it in the package.
3. **Permission Errors in S3**
    - **Problem**: While trying to read the JSON file from S3, permission-related errors occurred due to missing access rights.
    - **Solution**: I resolved this by assigning the **proper permissions** to the Lambda function through an **IAM role** that allowed it to access the S3 bucket and DynamoDB.
4. **Data Processing Logic**
    - **Problem**: Initially, I was able to process only basic user data fields (userID and timestamp), but more fields and multiple users were needed.
    - **Solution**: I enhanced the Lambda function logic to handle **additional fields** and multiple users by updating the JSON schema, dynamically parsing the incoming data, and writing each user's data to DynamoDB.

Conclusion

AWS Lambda, S3, and DynamoDB together provide a powerful, serverless infrastructure that is highly scalable, cost-efficient, and ideal for event-driven architectures. They allow for a high degree of automation and are well-suited for modern applications that need to scale dynamically. However, their limitations in execution time, cold starts, and query complexity make them less suitable for stateful applications or those with highly complex queries.

For applications requiring fast, scalable, event-driven processing, this architecture is a robust choice. However, if your application demands advanced querying, transactions, or stateful workflows, traditional databases such as **MongoDB** and **PostgreSQL** or other alternatives might be more appropriate.