

# AWS Lambda Function Deployment

## 1. Introduction

### Case Study Overview

This case study examines the deployment of an AWS Lambda function using Node.js 20.x, detailing the process of creating a zip file for the function code and integrating it with an S3 bucket and DynamoDB for data processing.

### Key Feature and Application

The unique feature of this case study is the use of AWS Lambda, a serverless compute service that allows users to run code without provisioning or managing servers. This enables quick scaling and cost efficiency. The practical application involves setting up a Lambda function that processes JSON files uploaded to an S3 bucket and stores the processed data in a DynamoDB table.

## 2. Step-by-Step Explanation

**Task 2 :Write a Lambda function in Node.js that triggers when a JSON file is uploaded to an S3 bucket.**

### Step 1: Initial Setup

#### Create an S3 Bucket

##### 1. Log in to the AWS Management Console:

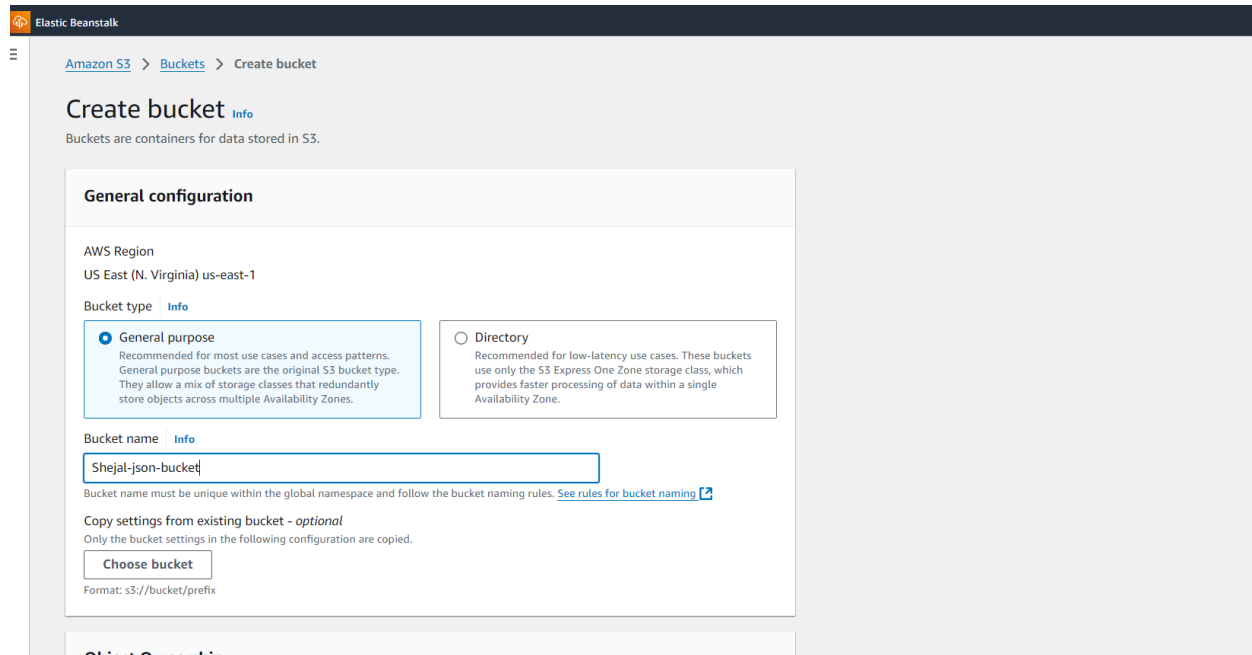
- Open a web browser and go to the [AWS Management Console](https://aws.amazon.com/console/).
- Enter your credentials to log in.

##### 2. Navigate to S3:

- In the AWS Management Console, type "S3" in the search bar and select S3 from the services list.

##### 3. Create a Bucket:

- Click on the Create bucket button.
- Enter the bucket name as `shejal-json-bucket`.
- Choose the AWS region where you want the bucket to be created.



Amazon S3 > Buckets > Create bucket

## Create bucket [Info](#)

Buckets are containers for data stored in S3.

### General configuration

AWS Region  
US East (N. Virginia) us-east-1

Bucket type [Info](#)

☒ **General purpose**  
Recommended for most use cases and access patterns. General purpose buckets are the original S3 bucket type. They allow a mix of storage classes that redundantly store objects across multiple Availability Zones.

☐ **Directory**  
Recommended for low-latency use cases. These buckets use only the S3 Express One Zone storage class, which provides faster processing of data within a single Availability Zone.

Bucket name [Info](#)

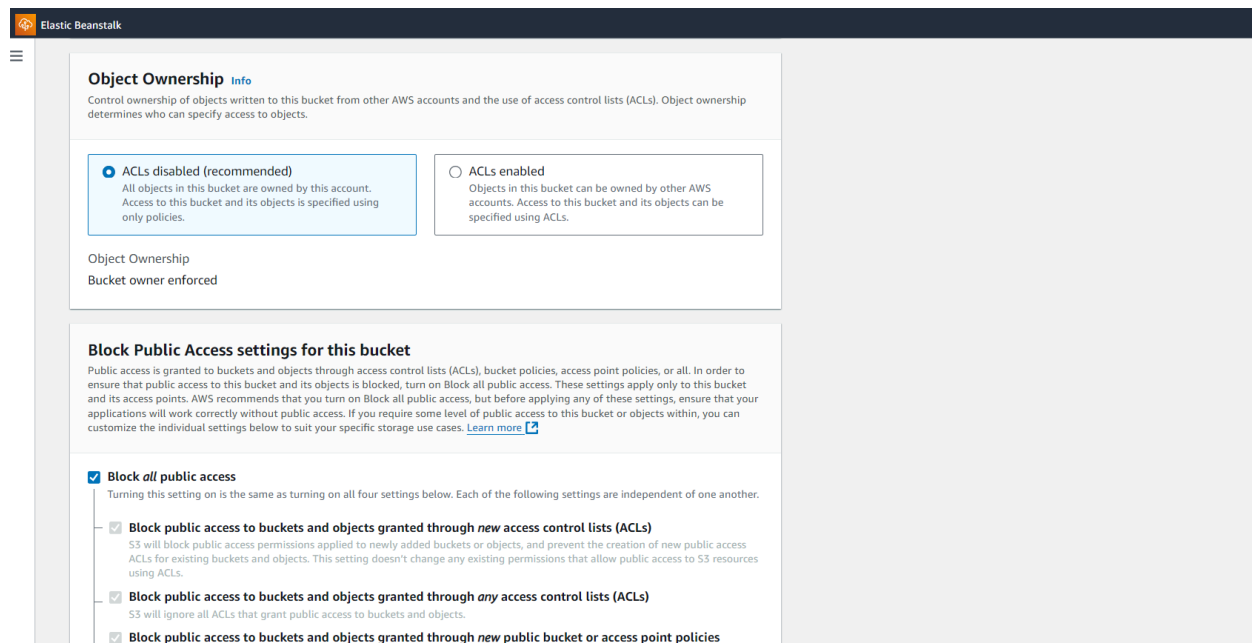
Bucket name must be unique within the global namespace and follow the bucket naming rules. [See rules for bucket naming](#)

Copy settings from existing bucket - *optional*  
Only the bucket settings in the following configuration are copied.

Format: s3://bucket/prefix

#### 4. Configure Bucket Settings:

- **Block Public Access Settings:** If you need public access (e.g., for testing purposes), uncheck the option that blocks all public access, but ensure you understand the security implications.
- **Bucket Versioning:** Enable versioning if you want to keep multiple versions of objects in the bucket.
- Click Create bucket.



### Object Ownership [Info](#)

Control ownership of objects written to this bucket from other AWS accounts and the use of access control lists (ACLs). Object ownership determines who can specify access to objects.

☒ **ACLs disabled (recommended)**  
All objects in this bucket are owned by this account. Access to this bucket and its objects is specified using only policies.

☐ **ACLs enabled**  
Objects in this bucket can be owned by other AWS accounts. Access to this bucket and its objects can be specified using ACLs.

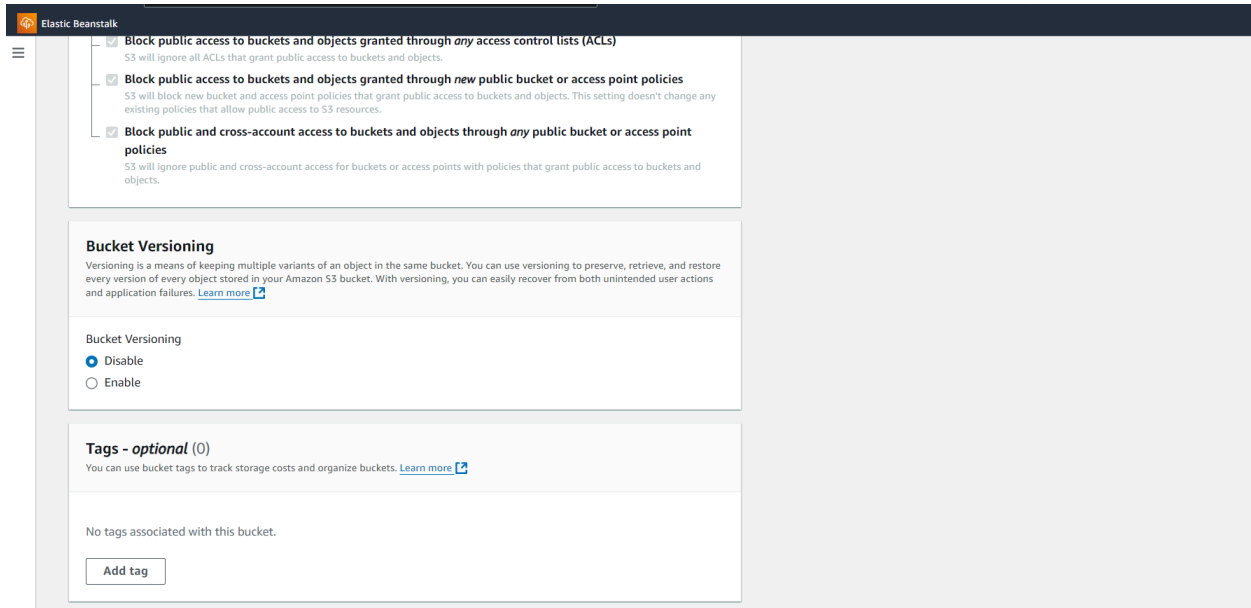
Object Ownership  
Bucket owner enforced

### Block Public Access settings for this bucket

Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to this bucket and its objects is blocked, turn on Block all public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, ensure that your applications will work correctly without public access. If you require some level of public access to this bucket or objects within, you can customize the individual settings below to suit your specific storage use cases. [Learn more](#)

☒ **Block all public access**  
Turning this setting on is the same as turning on all four settings below. Each of the following settings is independent of one another.

- ☒ **Block public access to buckets and objects granted through new access control lists (ACLs)**  
S3 will block public access permissions applied to newly added buckets or objects, and prevent the creation of new public access ACLs for existing buckets and objects. This setting doesn't change any existing permissions that allow public access to S3 resources using ACLs.
- ☒ **Block public access to buckets and objects granted through any access control lists (ACLs)**  
S3 will ignore all ACLs that grant public access to buckets and objects.
- ☒ **Block public access to buckets and objects granted through new public bucket or access point policies**

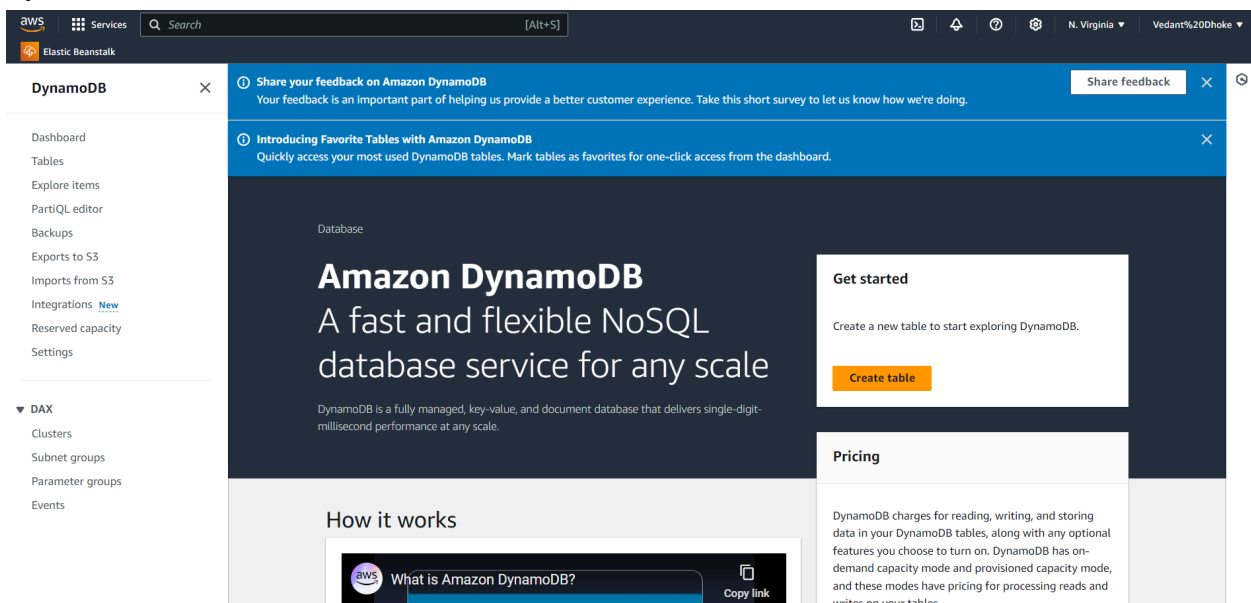


## Tasks1 : Create a DynamoDB table with relevant attributes.

### Create a DynamoDB Table

#### 1. Go to DynamoDB:

- In the AWS Management Console, type "DynamoDB" in the search bar and select DynamoDB from the services list.



#### 2. Create a Table:

- Click on the Create table button.
- Enter the table name as 'Shejal-ProcessedTable'.
- Set the primary key as 'userID' with the type set to String.

- Configure any additional settings, such as read/write capacity (you can start with the default settings).

The screenshot shows the 'Create table' page in the AWS Management Console. The breadcrumb navigation is 'DynamoDB > Tables > Create table'. The page title is 'Create table'. Under 'Table details', there is an 'Info' link and a note: 'DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.' The 'Table name' field is 'Shejal-ProcessedTable', with a note: 'This will be used to identify your table. Between 3 and 255 characters, containing only letters, numbers, underscores (\_), hyphens (-), and periods (.).' The 'Partition key' is 'userID' with a 'String' data type, and a note: '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.' The 'Sort key - optional' is 'timestamp' with a 'Number' data type, and a note: '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.'

**Table settings**

### 3. Create the Table:

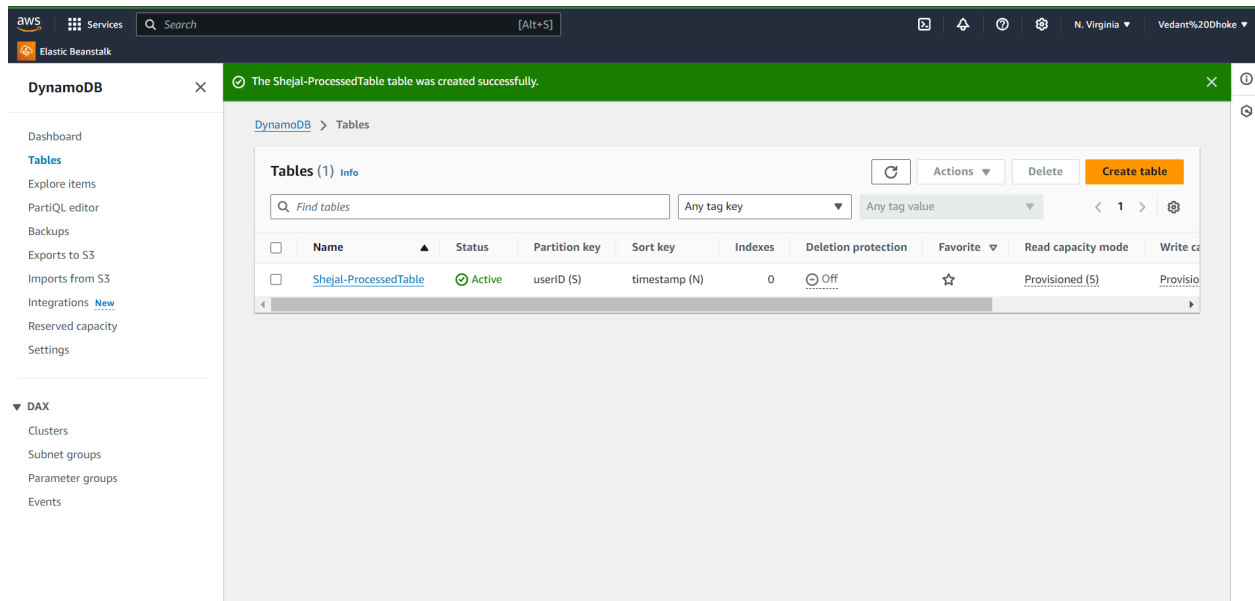
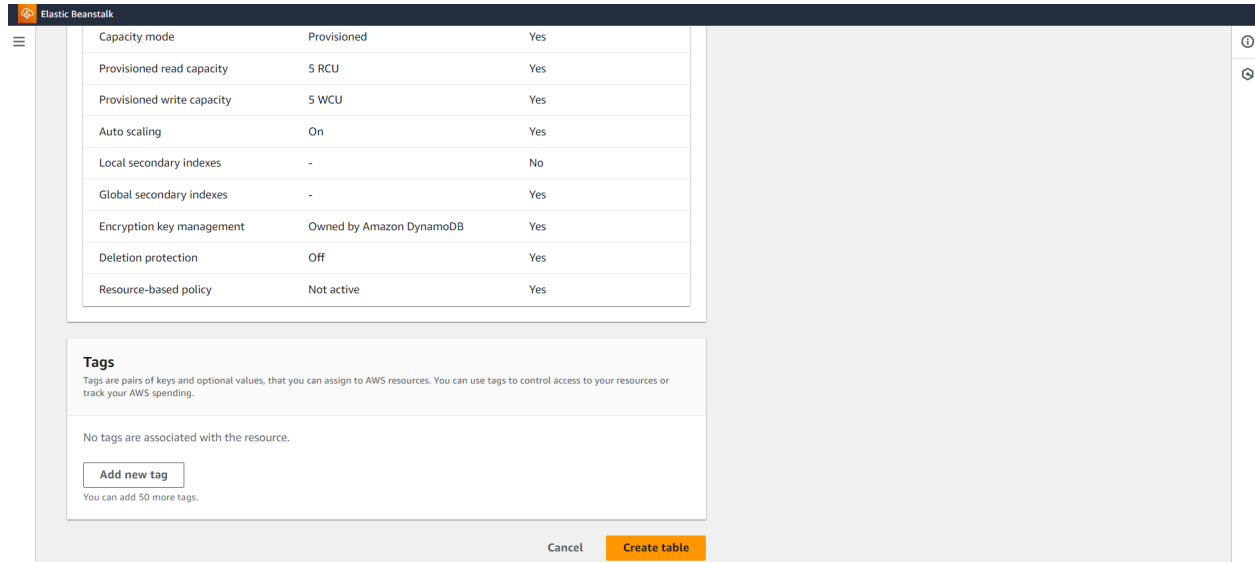
- Click Create to finish setting up your DynamoDB table.
- Wait for the table status to change to Active.

The screenshot shows the 'Table settings' section. There are two tabs: 'Default settings' (selected) and 'Customize settings'. The 'Default settings' tab has a description: 'The fastest way to create your table. You can modify these settings now or after your table has been created.' The 'Customize settings' tab has a description: 'Use these advanced features to make DynamoDB work better for your needs.'

**Default table settings**

These are the default settings for your new table. You can change some of these settings after creating the table.

| Setting                    | Value                    | Editable after creation |
|----------------------------|--------------------------|-------------------------|
| Table class                | DynamoDB Standard        | Yes                     |
| Capacity mode              | Provisioned              | Yes                     |
| Provisioned read capacity  | 5 RCU                    | Yes                     |
| Provisioned write capacity | 5 WCU                    | Yes                     |
| Auto scaling               | On                       | Yes                     |
| Local secondary indexes    | -                        | No                      |
| Global secondary indexes   | -                        | Yes                     |
| Encryption key management  | Owned by Amazon DynamoDB | Yes                     |
| Deletion protection        | Off                      | Yes                     |
| Resource-based policy      | Not active               | Yes                     |



## Set Up AWS Lambda

### 1. Navigate to Lambda:

- In the AWS Management Console, type "Lambda" in the search bar and select Lambda from the services list.

### 2. Create a New Function:

- Click on the Create function button.
- Select Author from scratch.
- Enter the function name as 'ProcessJsonData'.
- Choose the runtime as Node.js 20.x.

**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

**Permissions** [Info](#)  
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this

### 3. Set Permissions:

- Under Permissions, click on Change default execution role.
- Select Create a new role with basic Lambda permissions.
- After creating the role, you will need to add permissions for S3 and DynamoDB.
- Go to the IAM service in the AWS Management Console, find the role created for your Lambda function, and attach the following policies:

**Trusted entity type**

☒ **AWS service**  
Allow AWS services like EC2, Lambda, or others to perform actions in this account.

☐ **AWS account**  
Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.

☐ **Web identity**  
Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.

☐ **SAML 2.0 federation**  
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.

☐ **Custom trust policy**  
Create a custom trust policy to enable others to perform actions in this account.

---

**Use case**  
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

**Service or use case**

Choose a use case for the specified service.  
**Use case**  
☒ **Lambda**  
 Allows Lambda functions to call AWS services on your behalf.


- AmazonS3ReadOnlyAccess(to allow read access to S3)

**Permissions policies (3/959)** [Info](#)

Choose one or more policies to attach to your new role.

Filter by Type

Q AmazonS3ReadOnlyAccess X All types 1 match < 1 > ⚙

| <input checked="" type="checkbox"/> | Policy name <a href="#">?</a>  | Type        | Description                                    |
|-------------------------------------|--|-------------|--|
| <input checked="" type="checkbox"/> |  <a href="#">AmazonS3ReadOnlyAccess</a> | AWS managed | Provides read only access to all buckets vi... |

► **Set permissions boundary - optional**

Cancel Previous **Next**

- AmazonDynamoDBFullAccess (or create a custom policy with specific permissions to write to DynamoDB).


**Add permissions** [Info](#)

**Permissions policies (3/959)** [Info](#)

Choose one or more policies to attach to your new role.

Filter by Type

Q AmazonDynamoDBFullAccess X All types 1 match < 1 > ⚙

| <input checked="" type="checkbox"/> | Policy name <a href="#">?</a>  | Type        | Description                               |
|-------------------------------------|--|-------------|---|
| <input checked="" type="checkbox"/> |  <a href="#">AmazonDynamoDBFullAccess</a> | AWS managed | Provides full access to Amazon DynamoD... |

► **Set permissions boundary - optional**

CloudWatchLogsFullAccess (permission to log function execution and errors.)


**Add permissions** [Info](#)

**Permissions policies (3/959)** [Info](#)

Choose one or more policies to attach to your new role.

Filter by Type

Q CloudWatchLogsFullAccess X All types 1 match < 1 > ⚙

| <input checked="" type="checkbox"/> | Policy name <a href="#">?</a>  | Type        | Description                             |
|-------------------------------------|--|-------------|---|
| <input checked="" type="checkbox"/> |  <a href="#">CloudWatchLogsFullAccess</a> | AWS managed | Provides full access to CloudWatch Logs |

► **Set permissions boundary - optional**

Cancel Previous **Next**

The screenshot shows the AWS IAM console for a role. The **Summary** tab is active, displaying the role's ARN as `arn:aws:iam::022499016110:role/LambdaS3DynamoDBRole` and a maximum session duration of 1 hour. Below this, the **Permissions** tab is selected, showing a list of three managed policies attached to the role:

| Policy name                              | Type        | Attached entities |
|--|-------------|-------------------|
| <a href="#">AmazonDynamoDBFullAccess</a> | AWS managed | 1                 |
| <a href="#">AmazonS3ReadOnlyAccess</a>   | AWS managed | 2                 |
| <a href="#">CloudWatchLogsFullAccess</a> | AWS managed | 1                 |

#### 4. Configure the Lambda Function:

- In the Lambda function configuration, go to the Function code section and add your code (you can add this later after creating the zip file).

The screenshot shows the AWS Lambda console for a function. The **Code** tab is active, displaying the function's code in a text editor. The code is a JavaScript handler that returns a 200 status code and a JSON body containing the message "Hello from Lambda!".

```
1 export const handler = async (event) => {
2   // TODO Implement
3   const response = {
4     statusCode: 200,
5     body: JSON.stringify('Hello from Lambda!'),
6   };
7   return response;
8 };
9
```

#### Step 1: Create a Zip File for Your Lambda Function Code

1. Organize Your Code: Create a project directory on your local machine. For example:

...

```
Function/
├── index.js
└── package.json
```

...



index.js

```
const AWS = require("aws-sdk");
const S3 = new AWS.S3();
const DynamoDB = new AWS.DynamoDB.DocumentClient();

exports.handler = async (event) => {
  const bucketName = event.Records[0].s3.bucket.name;
  const objectKey = event.Records[0].s3.object.key;

  try {
    const params = {
      Bucket: "shejal-json-bucket",
      Key: "sample.json",
    };

    const data = await S3.getObject(params).promise();
    const json = JSON.parse(data.Body.toString());

    const dynamoParams = {
      TableName: "Shejal-ProcessedTable",
      Item: {
        userID: json.userID,
        timestamp: json.timestamp,
      },
    };

    await DynamoDB.put(dynamoParams).promise();
    return { status: "success" };
  } catch (err) {
    console.error(err);
    return { status: "error", error: err.message };
  }
};
```

2. Install Dependencies: If your Lambda function relies on external libraries (like `aws-sdk`), make sure to install them in the project directory:

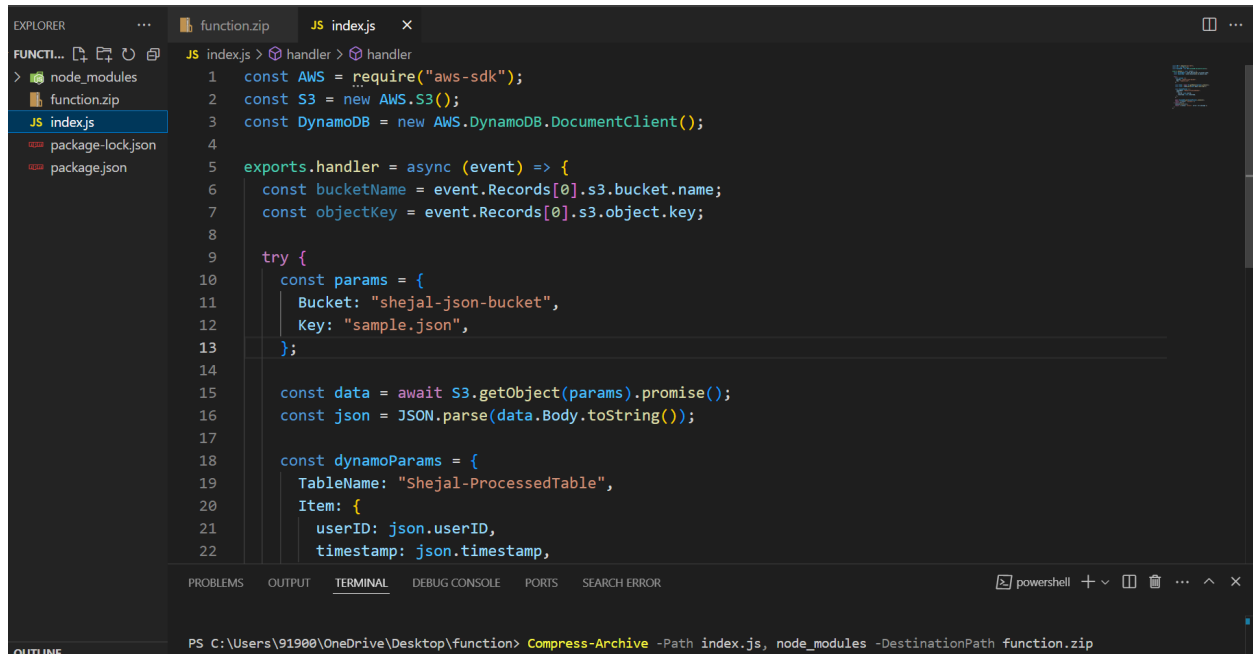
```
```bash
npm install aws-sdk
```
```

3. Create the Zip File:

- Navigate to your project directory.
- Use the following command to create a zip file containing your code and dependencies:

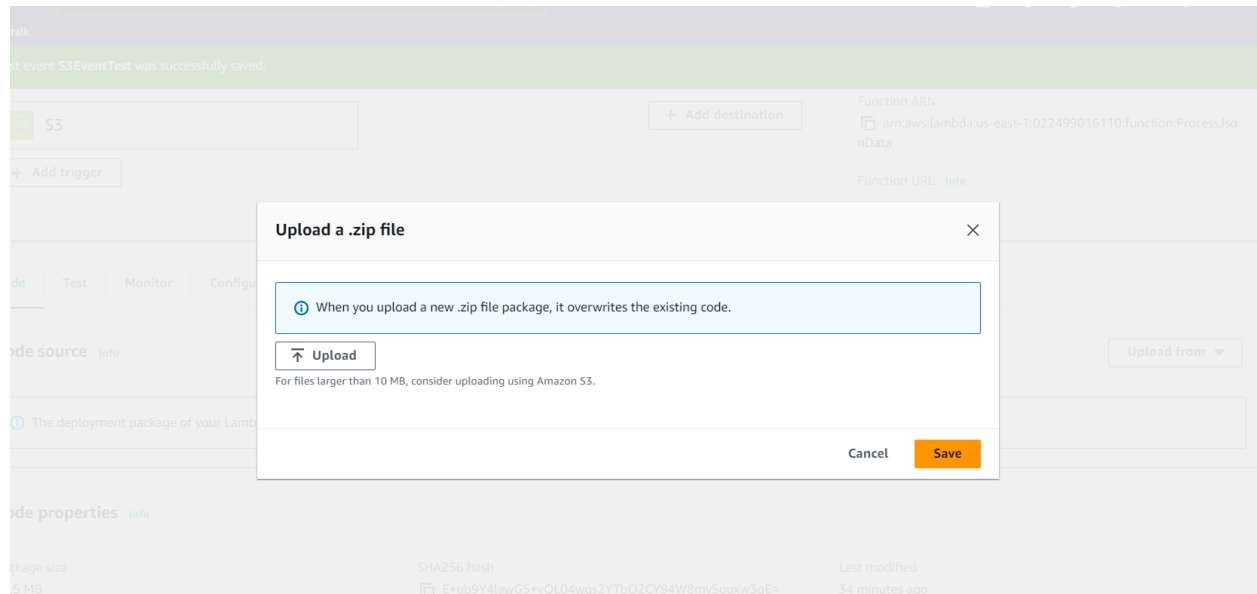
```
```bash
zip -r Function.zip .
```
```

This command includes all files in the current directory into the `Function.zip` archive.

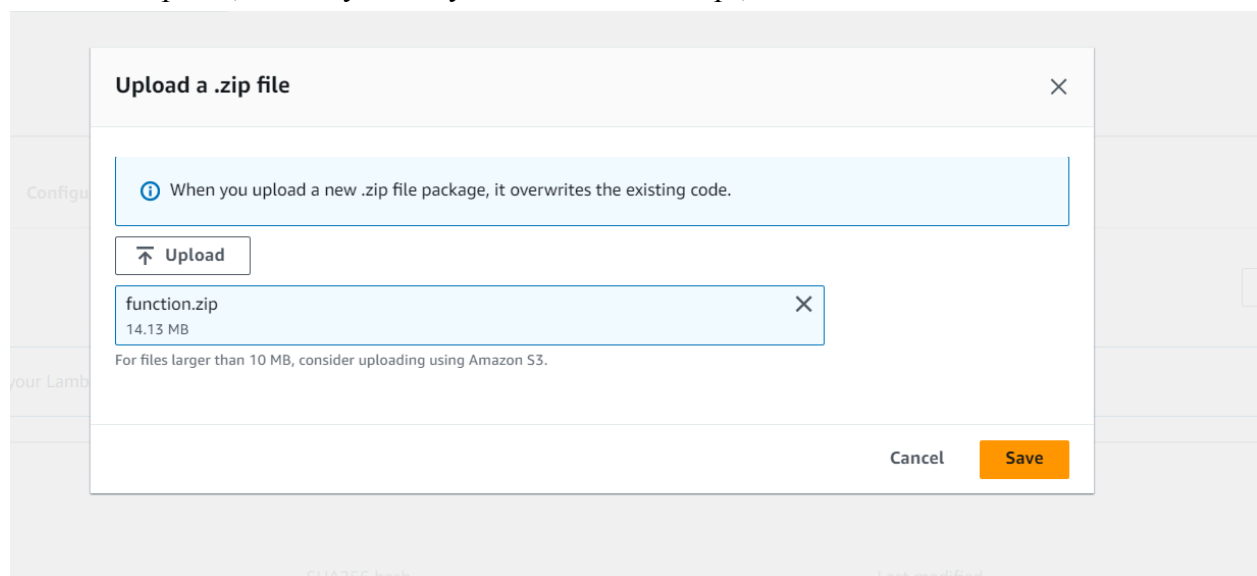


4. Upload the Zip File:

- Return to the AWS Lambda console, select your function, and under the "Code" section, choose "Upload from" > ".zip file."



- Click "Upload," select your `MyLambdaFunction.zip`, and then click "Save."



The screenshot shows the AWS Lambda console for the function 'ProcessJsonData'. The 'Code source' tab is selected, showing a message: 'The deployment package of your Lambda function "ProcessJsonData" is too large to enable inline code editing. However, you can still invoke your function.' Below this, the 'Code properties' section displays: Package size (13.5 MB), SHA256 hash (E+ub9Y4lawG5+vQL04wgs2YTb02CY94W8mv5ouxw3qE=), and Last modified (1 day ago). The 'Runtime settings' section shows: Runtime (Node.js 20.x), Handler (index.handler), and Architecture (x86\_64). There are buttons for 'Edit', 'Edit runtime management configuration', and 'Runtime management configuration'.

### Step 3: Add an S3 Trigger to Your Lambda Function

#### 1. Navigate to Your Lambda Function:

- In the AWS Management Console, type "Lambda" in the search bar and select **Lambda** from the services list.
- Click on your Lambda function (**ProcessJsonData**) to open its configuration page.

#### 2. Configure the Trigger:

- In the **Function overview** section, click on the + **Add trigger** button.

#### 3. Select the Trigger Type:

- From the dropdown list of triggers, select **S3**.

#### 4. Configure S3 Trigger Settings:

- **Bucket:** Choose your S3 bucket (e.g., **shejal-json-bucket**) from the dropdown list. If the bucket is not listed, ensure it is created and you have the necessary permissions to access it.
- **Event type:** Select **All object create events**. This option will trigger the Lambda function whenever a new object is created in the specified S3 bucket.
- **Prefix and Suffix (optional):** If you want to limit the trigger to specific files, you can set a prefix (e.g., **uploads/**) or suffix (e.g., **.json**). Otherwise, leave these fields blank.

The screenshot shows the AWS Elastic Beanstalk console's 'Trigger configuration' page. At the top, there's a navigation bar with the AWS logo, 'Services', a search bar, and a '[Alt+S]' shortcut. Below this, the 'Elastic Beanstalk' logo is visible. The main content area is titled 'Trigger configuration' with an 'info' link. It features a dropdown menu for the trigger type, currently set to 'S3', with sub-labels 'aws', 'asynchronous', and 'storage'. Under the 'Bucket' section, a text input field contains 's3/shejal-json-bucket', accompanied by a search icon, a clear button (X), and a refresh button (circular arrow). Below the input, it states 'Bucket region: us-east-1'. The 'Event types' section includes a description and a dropdown menu. Below this, there are four buttons: 'All object create events' (with an X), 'PUT' (with an X), 'POST' (with an X), and 'COPY' (with an X). A fifth button, 'Multipart upload completed' (with an X), is shown below these. The 'Prefix - optional' section has a description and a text input field containing 'json/'. The 'Suffix - optional' section also has a description and a text input field containing '.json'.

##### 5. Add the Trigger:

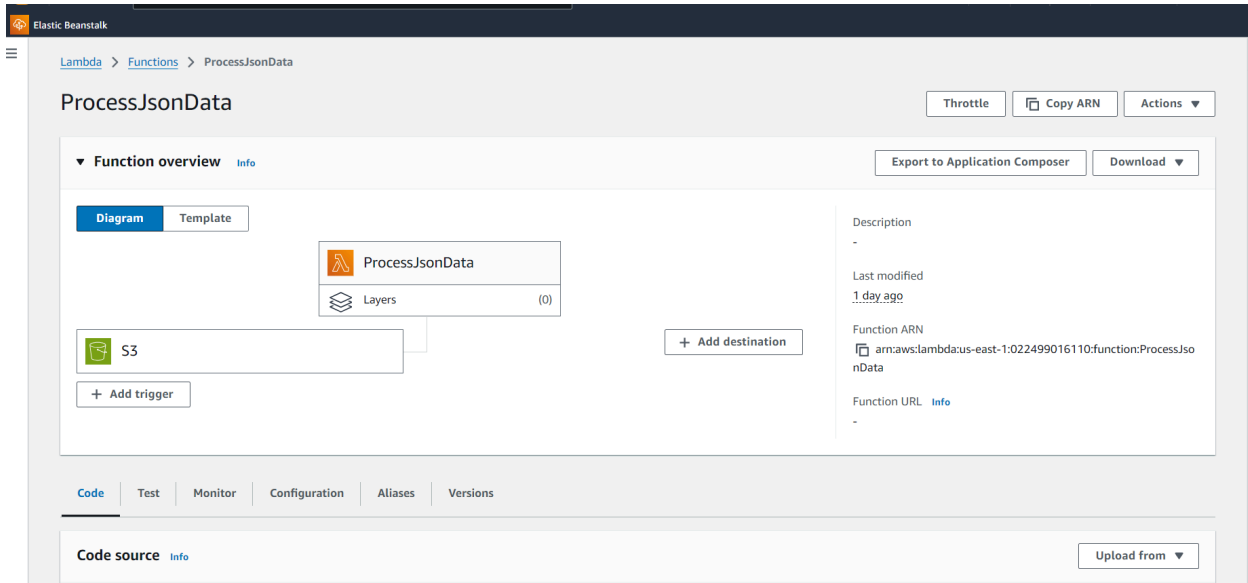
- After configuring the trigger settings, click the **Add** button at the bottom of the configuration panel.

##### 6. Review the Trigger Configuration:

- After adding the trigger, it should appear in the **Configuration** tab under the **Triggers** section. Ensure that it shows the correct bucket and event type.
- You may also see a notification that your Lambda function's execution role has been updated to allow access to the specified S3 bucket.

##### 7. Save Changes:

- Although the trigger is added, it's a good practice to click the **Save** button at the top right corner to ensure all configurations are updated properly.



## Step 4: Test the Lambda Function

### 1. Test the Function:

- Click on the **Test** tab within the Lambda function console.
- If you haven't created a test event yet, you'll need to create one:
  - Click **Configure test event**.
  - Select **Create new test event**.

Name the event (e.g., TestEvent) and use the following JSON structure for the test event:

json

Copy code

```
{
  "Records": [
    {
      "s3": {
        "bucket": {
          "name": "shejal-json-bucket"
        },
        "object": {
          "key": "sample.json"
        }
      }
    }
  ]
}
```

- 
- Click **Create** to save the test event.
- Now, click the **Test** button to run the function with the test event. Check the **Execution result** and **Logs** to confirm the function executed without errors.

**Test event** [Info](#) Delete CloudWatch Logs Live Tail Save Test

To invoke your function without saving an event, modify the event, then choose Test. Lambda uses the modified event to invoke your function, but does not overwrite the original event until you choose Save changes.

Test event action

☐ Create new event ☒ Edit saved event

Event name

SSEventTest

**Event JSON** Format JSON

```
1 {
2   "Records": [
3     {
4       "s3": {
5         "bucket": {
6           "name": "shejal-json-bucket"
7         },
8         "object": {
9           "key": "sample.json"
10        }
11      }
12    ]
13  }
14 }
```

✓ Executing function: succeeded ([logs 2](#))

▼ Details

The area below shows the last 4 KB of the execution log.

```
{
  "status": "success"
}
```

**Summary**

|  |                      |
|--|----------------------|
| Code SHA-256                                 | Execution time       |
| E+ub9Y4lawG5+vQL04wgs2YTb02CY94W8mvSouxw3qE= | 13 seconds ago       |
| Request ID                                   | Function version     |
| db38ec9e-d530-4387-938c-34a4afd1bf75         | \$LATEST             |
| Init duration                                | Duration             |
| 700.26 ms                                    | 1550.70 ms           |
| Billed duration                              | Resources configured |
| 1551 ms                                      | 128 MB               |
| Max memory used                              |                      |
| 89 MB  |                      |

**Log output**

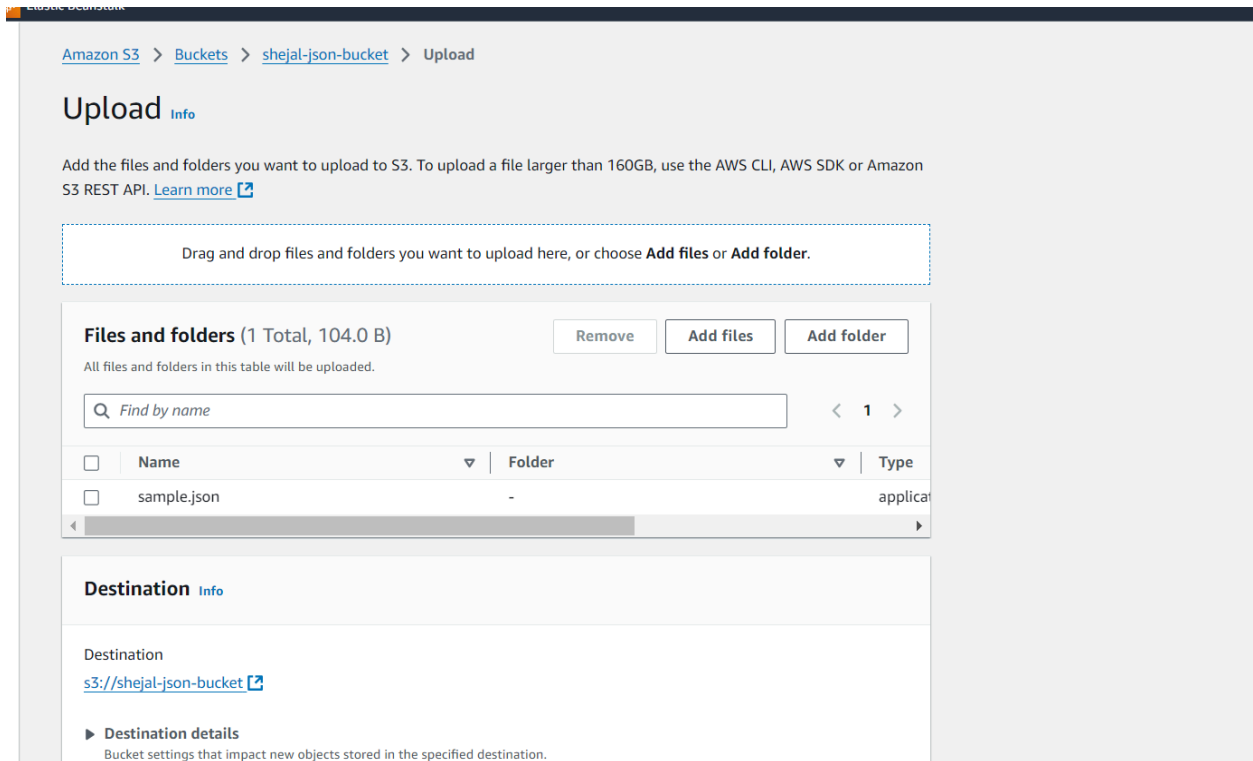
The section below shows the logging calls in your code. [Click here](#) to view the corresponding CloudWatch log group.

```
START RequestId: db38ec9e-d530-4387-938c-34a4afd1bf75 Version: $LATEST
END RequestId: db38ec9e-d530-4387-938c-34a4afd1bf75
REPORT RequestId: db38ec9e-d530-4387-938c-34a4afd1bf75 Duration: 1550.70 ms Billed Duration: 1551 ms Memory Size: 128 MB Max Memory Used: 89 MB Init Duration: 700.26 ms
```

## Step 5: Upload a Sample JSON File to S3

### 1. Navigate to Your S3 Bucket:

- In the AWS Management Console, type "S3" in the search bar and select **S3** from the services list.
- Click on your bucket named shejal-json-bucket to open it.



### 2. Upload a Sample JSON File:

- Click on the **Upload** button to begin the upload process.

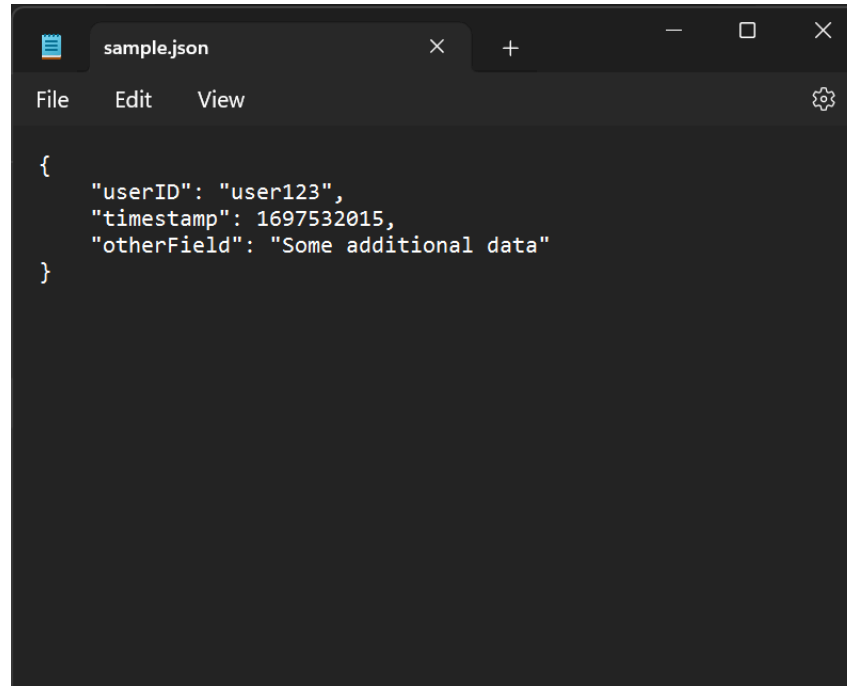
In the upload dialog, click **Add files** and select the sample JSON file from your local system. If you don't have the file yet, create one with the following structure and save it as sample.json:

json

Copy code

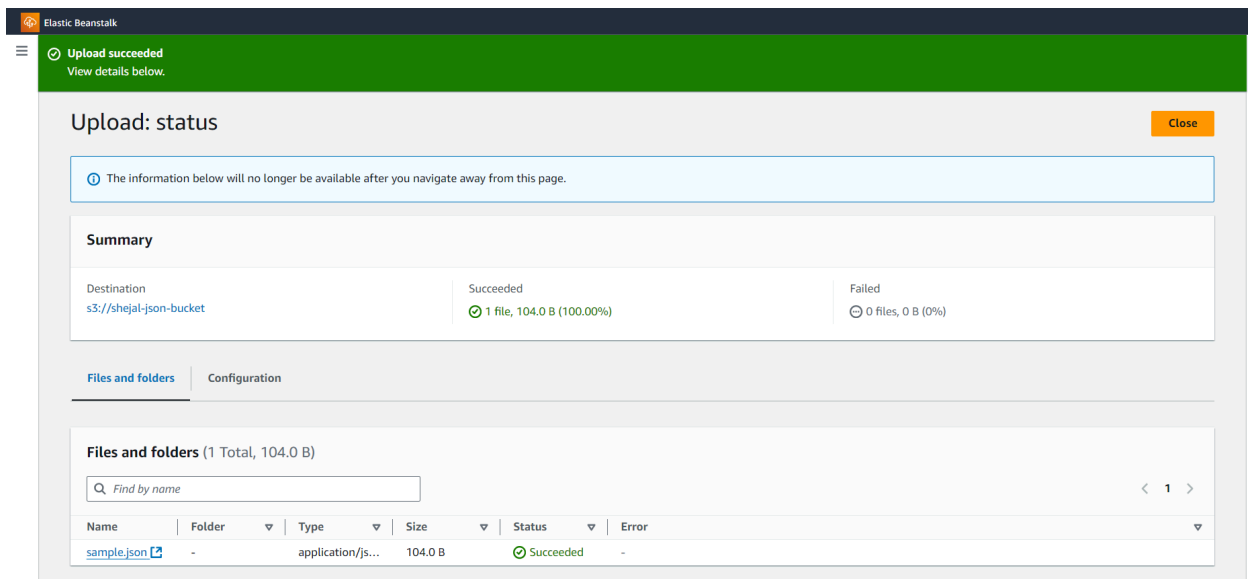
```
{
  "userID": "user123",
  "timestamp": 1697532015,
  "otherField": "Some additional data"
}
```





```
{
  "userID": "user123",
  "timestamp": 1697532015,
  "otherField": "Some additional data"
}
```

- After selecting the file, click **Upload** at the bottom of the dialog.



Upload: status

Upload succeeded  
View details below.

The information below will no longer be available after you navigate away from this page.

**Summary**

|  |  |                               |
|--|--|-------------------------------|
| Destination<br>s3://shejal-json-bucket | Succeeded<br>✔ 1 file, 104.0 B (100.00%) | Failed<br>✖ 0 files, 0 B (0%) |
|--|--|-------------------------------|

**Files and folders** | Configuration

Files and folders (1 Total, 104.0 B)

Find by name

| Name                        | Folder | Type              | Size    | Status      | Error |
|-----------------------------|--------|-------------------|---------|-------------|-------|
| <a href="#">sample.json</a> | -      | application/js... | 104.0 B | ✔ Succeeded | -     |

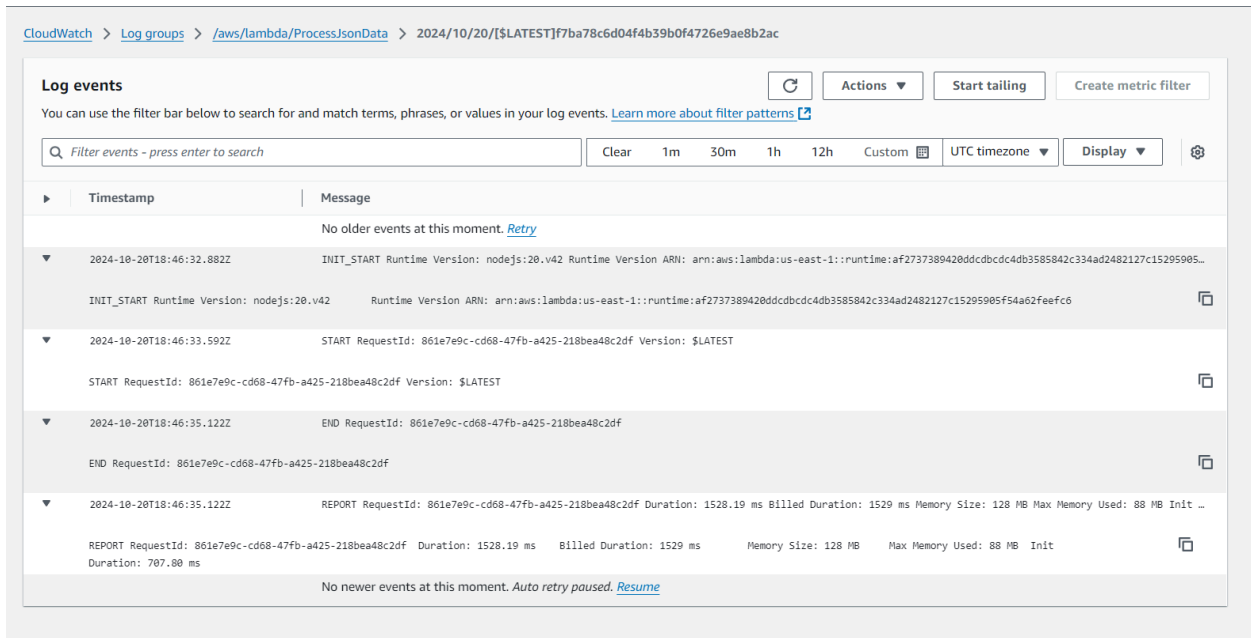
### 3. Verify the Upload:

- Once the upload is complete, you should see **sample.json** listed in your S3 bucket contents.
- You can click on the file to view its properties and confirm it was uploaded successfully.

#### 4. Check Lambda Execution:

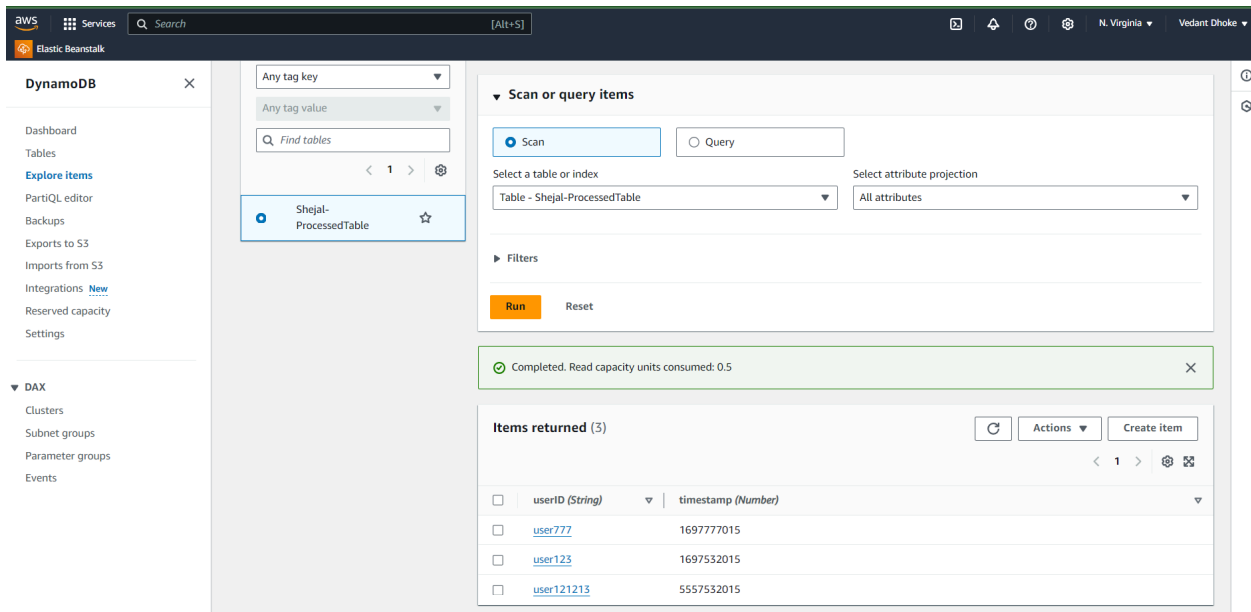
- After uploading the file, check the **Monitoring** tab of your Lambda function to ensure it was triggered successfully.
- Look for any logs generated in the **Log streams** section of CloudWatch to debug or confirm the operation of your Lambda function.

The screenshot displays the AWS CloudWatch console interface. The left sidebar shows the navigation menu with options like Dashboards, Alarms, Logs, Metrics, X-Ray traces, Events, Application Signals, Network monitoring, and Insights. The main content area is titled '/aws/lambda/ProcessJsonData' and shows the 'Log group details' for the log group 'arn:aws:logs-us-east-1:022499016110:log-group:/aws/lambda/ProcessJsonData:'. The details include Log class (Standard), ARN, Creation time (1 hour ago), Retention (Never expire), Stored bytes (-), Metric filters (0), Subscription filters (0), Contributor Insights rules (-), KMS key ID (-), Anomaly detection (Configure), Data protection (-), and Sensitive data count (-). Below the details, the 'Log streams' tab is selected, showing a list of 8 log streams. The log streams are listed with their names and last event times. The log stream names are: 2024/10/19/[LATEST]1d44fc9d025541afa088c7cfd8ba6b99, 2024/10/19/[LATEST]ce3a5bf460e94b8b952ea4540ae9103d, 2024/10/19/[LATEST]79c2c99e1ce1404aba33e5073b3830f1, 2024/10/19/[LATEST]5e02f75e668d40f2accfb3b1dc4a6c29, 2024/10/19/[LATEST]6c6c4a4014534ab299b65a46d66d3f3f, 2024/10/19/[LATEST]211d578d13b44a59a2f4553d7c433da6, 2024/10/19/[LATEST]547e0180cae44132ae3eb4cf820d428f, and 2024/10/19/[LATEST]2108e610fbeb4e09b59bfabb0d70cfd. The last event times are: 2024-10-19 12:26:41 (UTC), 2024-10-19 12:19:28 (UTC), 2024-10-19 12:13:46 (UTC), 2024-10-19 11:38:48 (UTC), 2024-10-19 11:33:28 (UTC), 2024-10-19 11:23:24 (UTC), 2024-10-19 11:19:48 (UTC), and 2024-10-19 11:17:53 (UTC).



Step 6: Verify Data in DynamoDB

- 1. Go to the DynamoDB console.
- 2. Navigate to your ProcessedData table.
- 3. Check if the data has been correctly written with the userID and timestamp.



**Guidelines and Best Practices**

**Keep Your Code Clean:** Regularly refactor your code to maintain clarity and efficiency.

**Test Locally:** Before zipping and uploading, test your Lambda function code locally to catch any issues early.

**Version Control:** Use version control systems like Git to track changes to your codebase, making it easier to manage updates.

**Dependency Management:** Only include necessary libraries in your zip file to reduce size and improve performance.

**Demonstration Preparation****Key Points**

1. **Overview of the Project:**
  - Briefly introduce the purpose of the project and the AWS services used (S3, Lambda, DynamoDB).
  - Explain the significance of automating data processing and the benefits of a serverless architecture.
2. **Architecture Explanation:**
  - Present a simple diagram illustrating the workflow: S3 bucket → Lambda function → DynamoDB.
  - Highlight how data flows through these services upon file upload.
3. **Step-by-Step Process:**
  - Demonstrate how to upload a JSON file to the S3 bucket.
  - Show how the Lambda function is triggered by the S3 event notification.
  - Verify that the data is successfully inserted into the DynamoDB table.
4. **Error Handling and Logging:**
  - Discuss how CloudWatch logs can be used to monitor function execution and troubleshoot issues.
  - Provide examples of common errors and how they were resolved during development.
5. **Security and Permissions:**
  - Explain the IAM roles and policies associated with the Lambda function.
  - Discuss the importance of granting the minimum necessary permissions for security.

**Potential Questions:****1. What challenges did you face during implementation?**

- **Permission Errors:**

- During the initial setup, I encountered permission errors while trying to access the S3 bucket and write to DynamoDB. This was resolved by ensuring that the execution role associated with the Lambda function had the necessary permissions (`s3:GetObject`, `s3:ListBucket`, and `dynamodb:PutItem`). I modified the IAM policy attached to the role to include these permissions.
- **Data Format Discrepancies:**
  - I faced issues with the JSON format in the uploaded files. If the JSON structure did not match the expected schema, the Lambda function would throw parsing errors. I implemented a validation step within the Lambda function to check the incoming data format and log errors for any discrepancies.
- **Lambda Execution Timeouts:**
  - Initially, my Lambda function was timing out due to the processing of larger files. To address this, I optimized the code to handle data more efficiently and increased the timeout setting for the function to accommodate larger file uploads.

## 2. How does this solution scale with increasing data volume?

- **AWS Lambda Scalability:**
  - AWS Lambda automatically scales based on the number of incoming requests. When multiple JSON files are uploaded to the S3 bucket simultaneously, Lambda can run multiple instances of the function concurrently, allowing it to process each file in parallel without manual intervention.
- **DynamoDB Scalability:**
  - DynamoDB is designed to scale horizontally. It can handle large amounts of data and high request rates by automatically distributing data across multiple partitions. With features like on-demand capacity mode, DynamoDB can adjust its throughput automatically based on the incoming traffic, ensuring consistent performance regardless of data volume.

## 3. What additional features could be added in the future?

- **AWS Step Functions:**
  - I could integrate AWS Step Functions to manage complex workflows, allowing for orchestrating multiple Lambda functions and handling tasks like retries and error handling seamlessly.
- **API Gateway:**
  - Adding an API Gateway would allow external applications to interact with the Lambda function via RESTful APIs, making it easier to upload files and retrieve data programmatically.
- **Amazon SNS for Notifications:**

- Implementing Amazon SNS (Simple Notification Service) could enhance monitoring by sending notifications upon successful data insertion into DynamoDB or alerting for any errors during processing, thus improving the observability of the system.

#### 4. How did you ensure data integrity and security?

- **Data Validation:**
  - I implemented validation logic within the Lambda function to ensure that the incoming JSON data met the expected format and contained all required fields (like `userID`). If the validation fails, the function logs the error and skips processing that file.
- **IAM Policies:**
  - The execution role for the Lambda function is configured with the principle of least privilege, ensuring that it only has access to the S3 bucket and DynamoDB table it needs. Regular audits of IAM policies are conducted to ensure that no excessive permissions are granted.

#### 5. Can you explain the cost implications of using these AWS services?

- **Amazon S3 Costs:**
  - S3 charges are based on the amount of data stored, the number of requests made, and the data transfer out of the S3 bucket. It's cost-effective for storing large amounts of data since pricing decreases with increased storage volume.
- **AWS Lambda Costs:**
  - Lambda pricing is based on the number of requests and the duration of the execution. There is a free tier that allows for a generous number of requests and compute time each month, which is beneficial for low-traffic applications.
- **DynamoDB Costs:**
  - DynamoDB has a pricing model based on data storage, read and write requests, and optional features like backups and data transfer. The on-demand capacity mode is flexible for varying workloads, ensuring I only pay for what I use.

## Conclusion

In this case study, we explored the integration of AWS services to create an automated data processing pipeline using Amazon S3, AWS Lambda, and DynamoDB. By setting up an S3 bucket to store JSON files, we configured a Lambda function that triggers upon file uploads. This function processes the incoming data and stores it in a DynamoDB table, enabling seamless data management and retrieval.

Key takeaways from this project include:

1. **Automation:** The use of S3 event notifications to trigger the Lambda function allows for real-time data processing without manual intervention, showcasing the power of serverless architectures.
2. **Scalability:** AWS services like Lambda and DynamoDB offer scalable solutions that can handle varying amounts of data, making it suitable for projects of any size.
3. **Error Handling and Debugging:** By monitoring CloudWatch logs, we identified potential issues and improved the function's reliability. This highlights the importance of logging and error handling in serverless applications.
4. **Hands-on Experience:** This project provided valuable experience in configuring AWS services, understanding IAM roles and permissions, and developing serverless applications, which are critical skills in today's cloud-centric development environment.

Overall, this case study demonstrates the effectiveness of leveraging AWS services to create efficient, automated workflows that enhance data management capabilities. Future enhancements could include adding more robust error handling, optimizing data processing, or integrating additional AWS services for advanced analytics.