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.

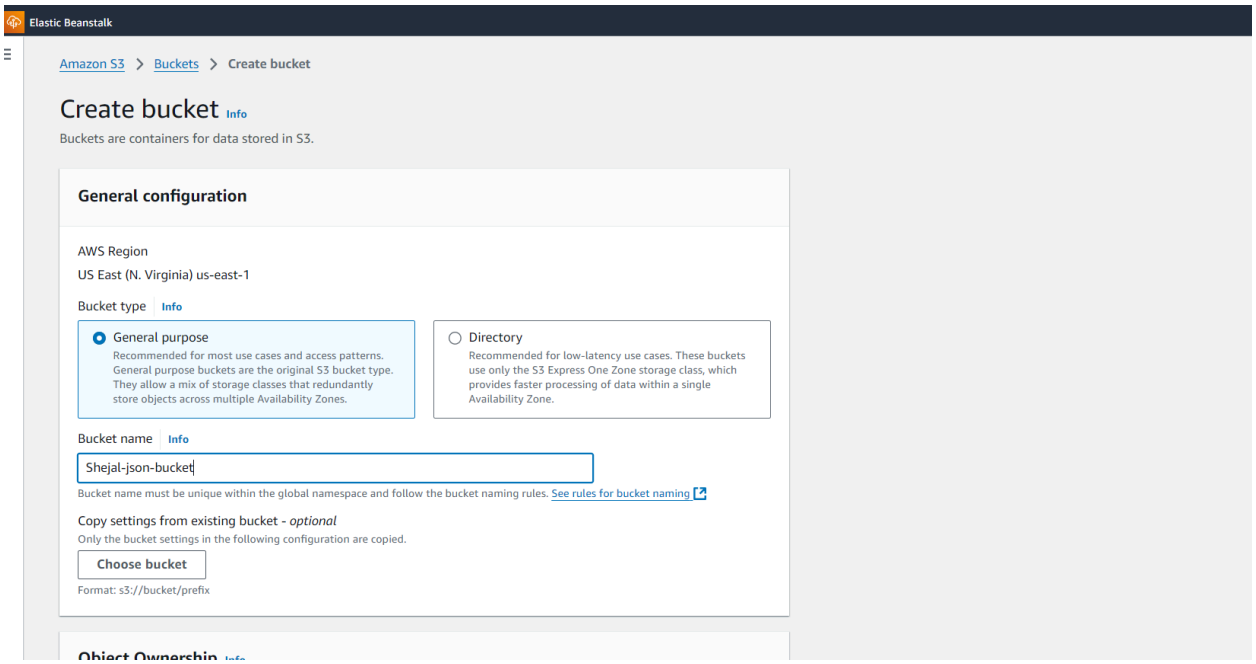
2. Step-by-Step Explanation
Task 2 :Write a Lambda function in Node.js that triggers when a JSON file is uploaded to an S3 bucket.

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.

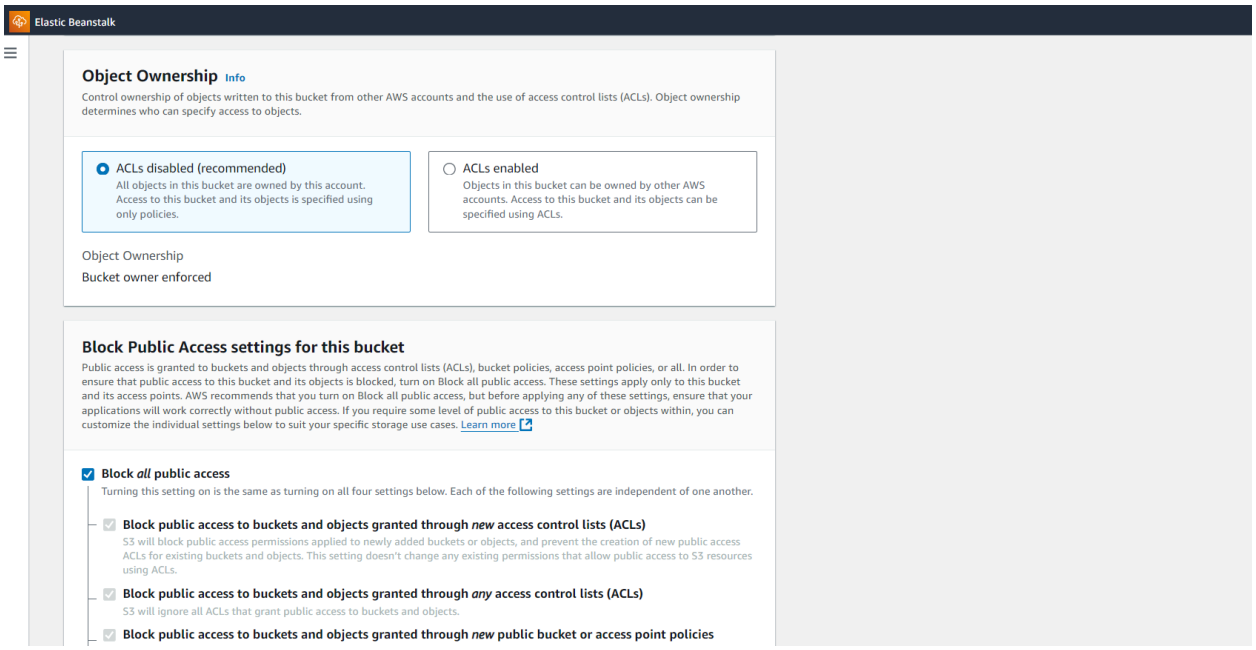
2. Navigate to S3:
  - In the AWS Management Console, type "S3" in the search bar and select S3 from the services list.

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.

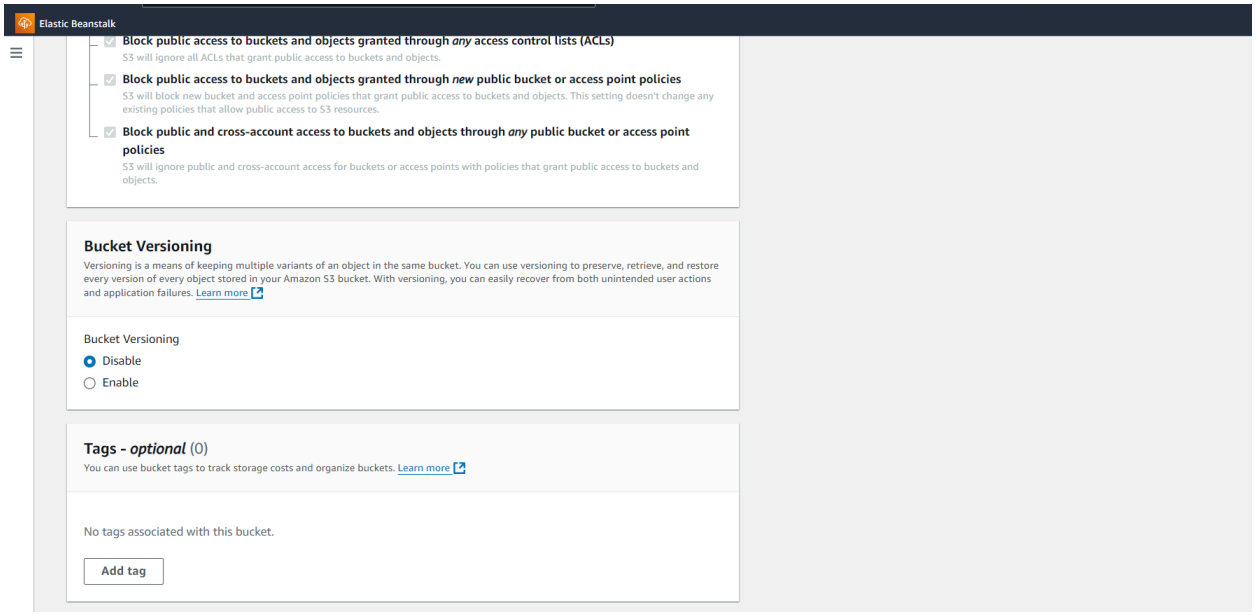


4. Configure Bucket Settings:

- 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.



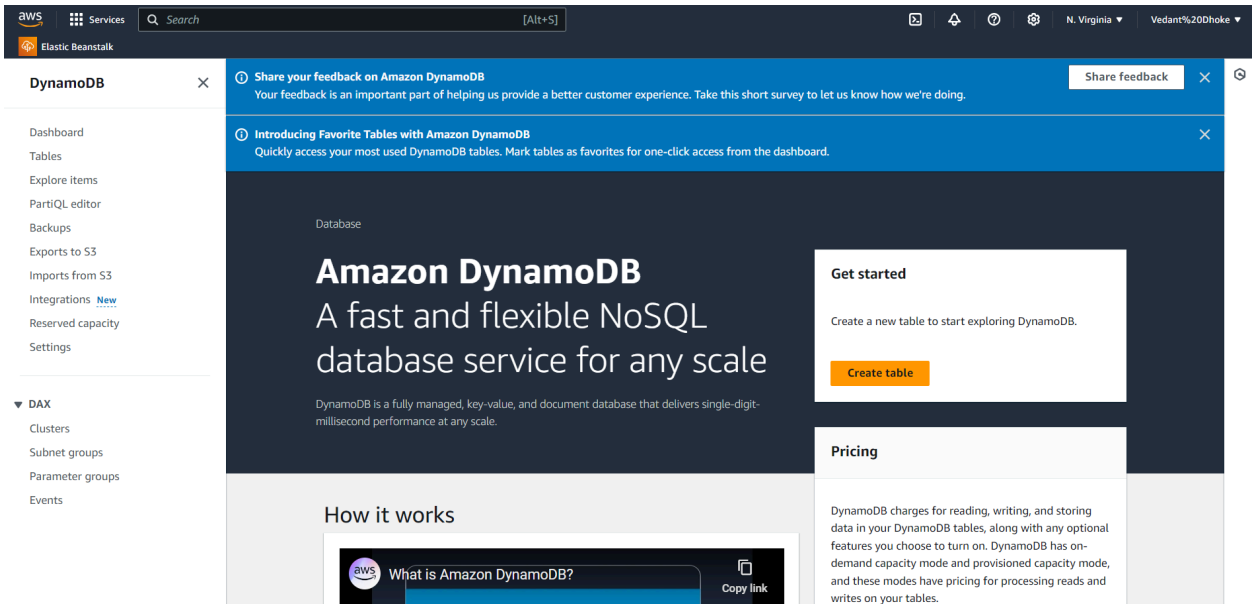
NAME: SHEJAL TIWARI DIV/ROLL NO: D15C/57



Tasks1 : Create a DynamoDB table with relevant attributes.

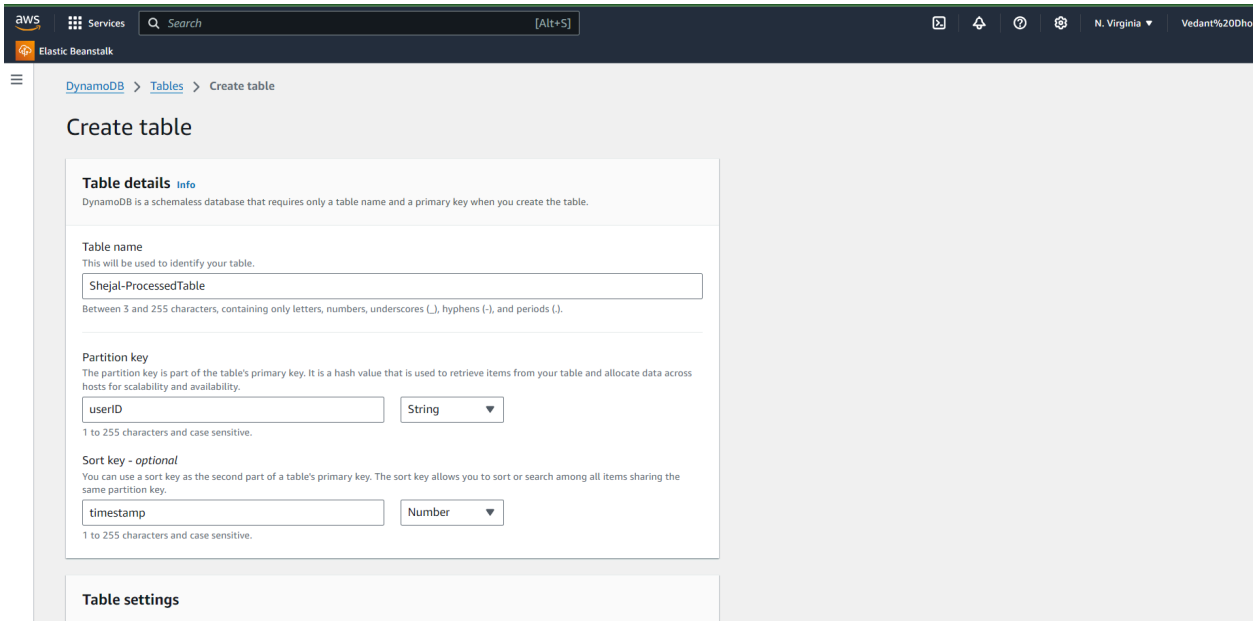
Create a DynamoDB Table

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



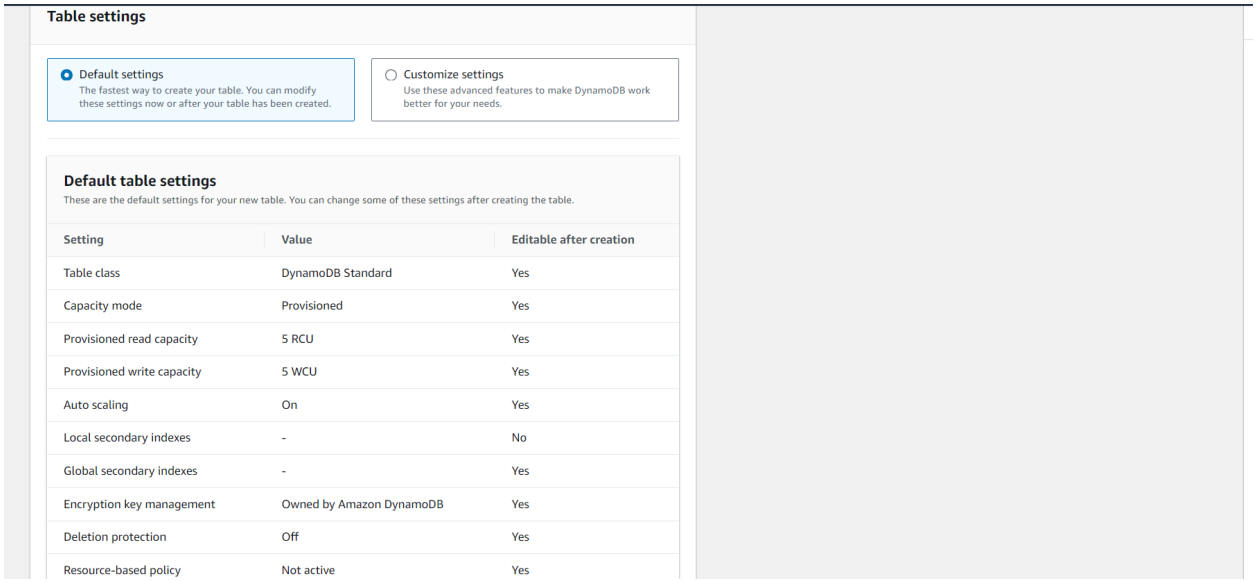
2. Create a Table:
   - Click on the Create table button.
   - Enter the table name as `Shejal-ProcessedTable`.
   - Set the primary key as `userID` with the type set to String.

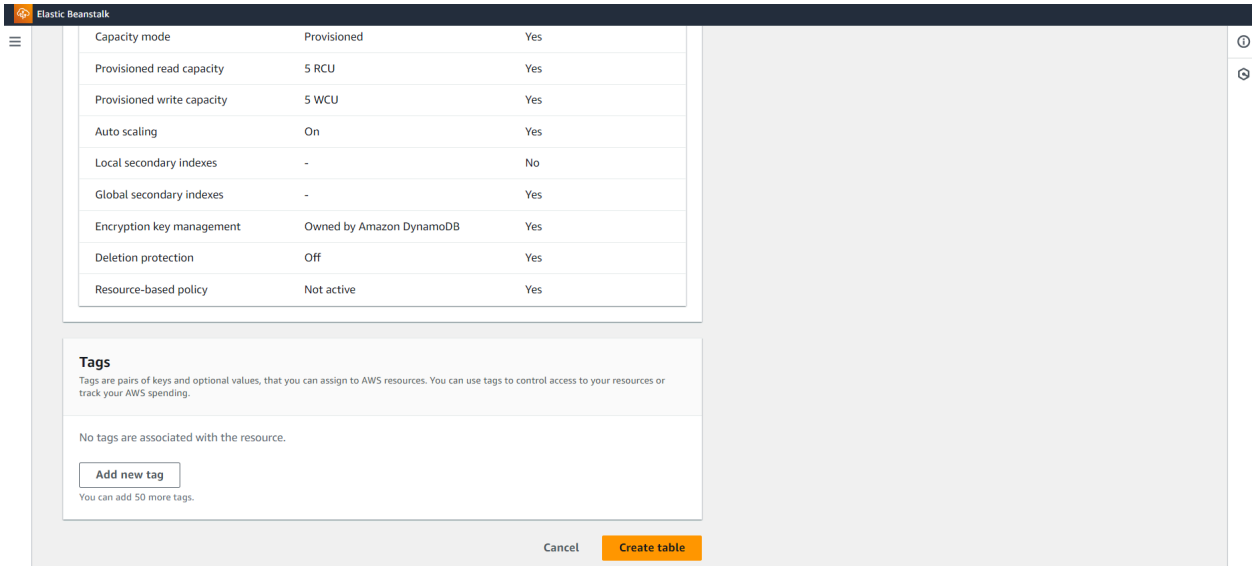
- Configure any additional settings, such as read/write capacity (you can start with the default settings).

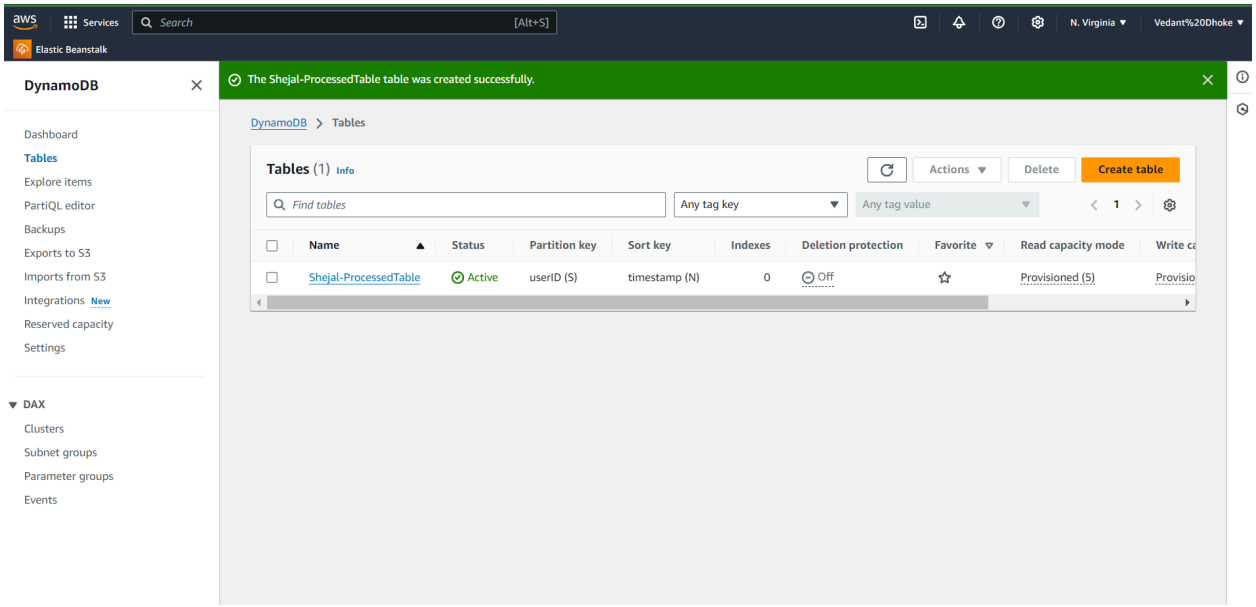


3. Create the Table:

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



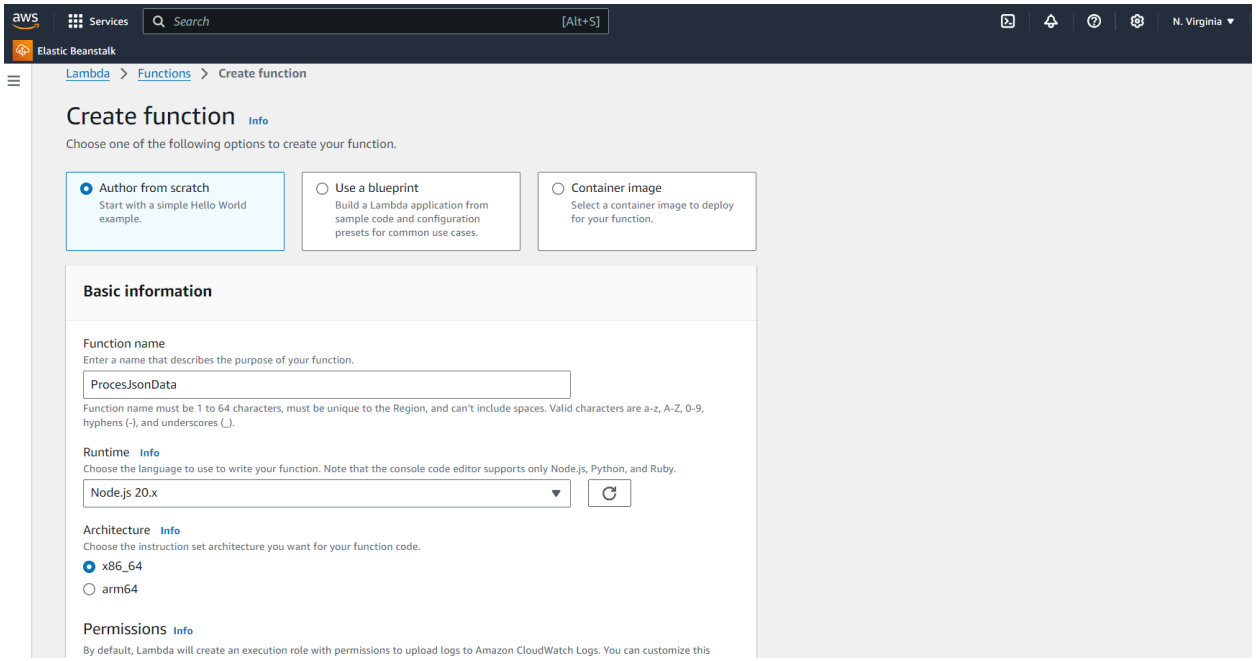




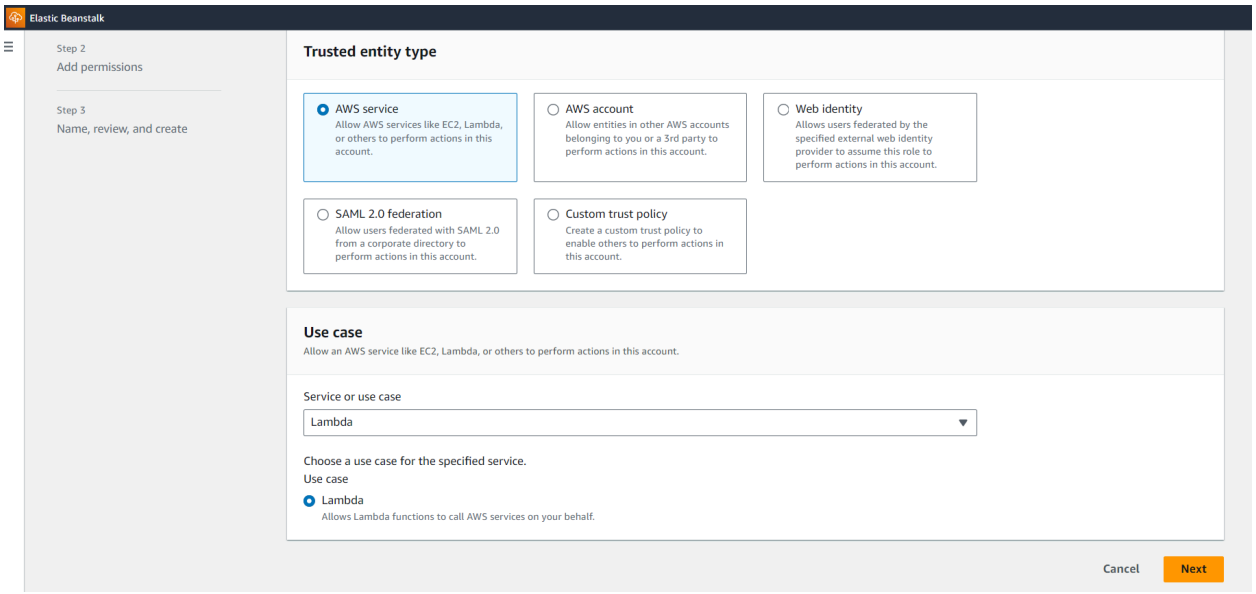
Set Up AWS Lambda

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

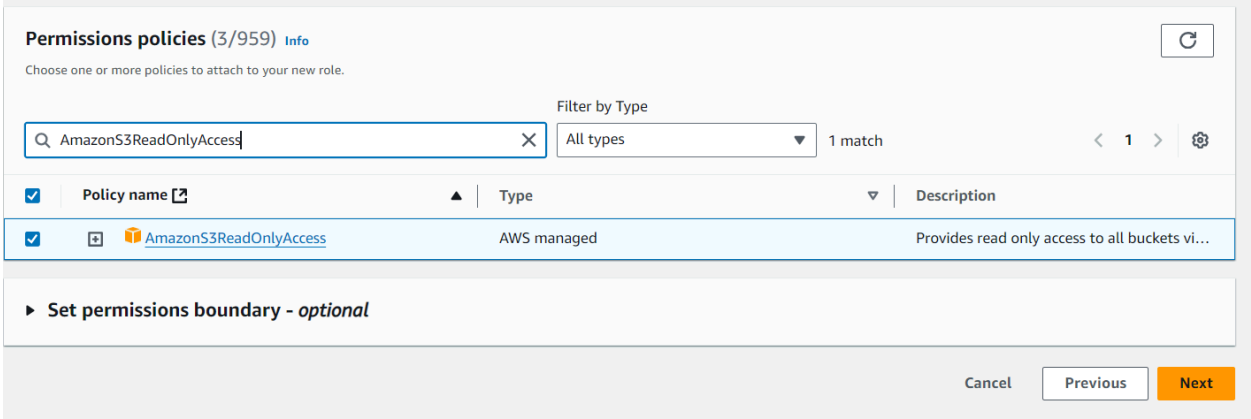
2. Create a New Function:
   - Click on the Create function button.
   - Select Author from scratch.
   - Enter the function name as `ProcessJsonData`.
   - Choose the runtime as Node.js 20.x.



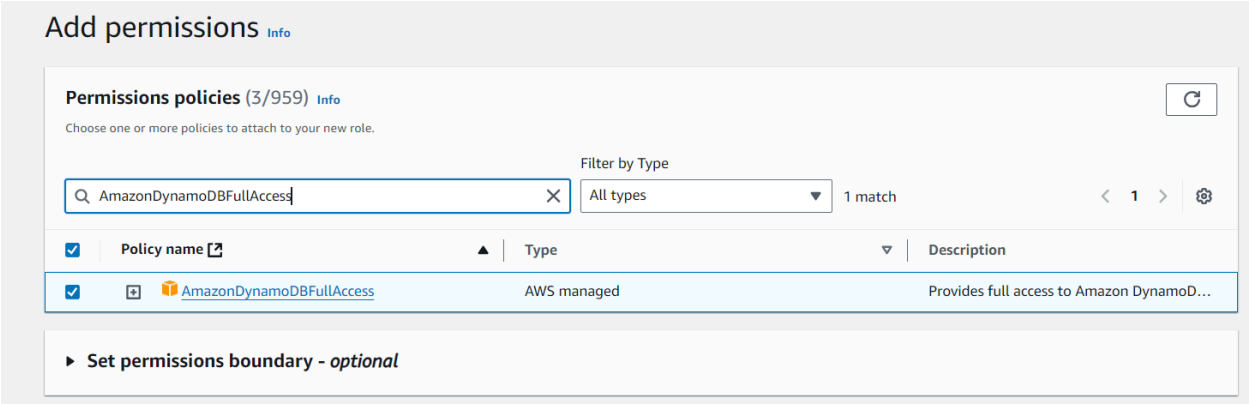
3. Set Permissions:
   - Under Permissions, click on Change default execution role.
   - Select Create a new role with basic Lambda permissions.
   - After creating the role, you will need to add permissions for S3 and DynamoDB.
   - Go to the IAM service in the AWS Management Console, find the role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



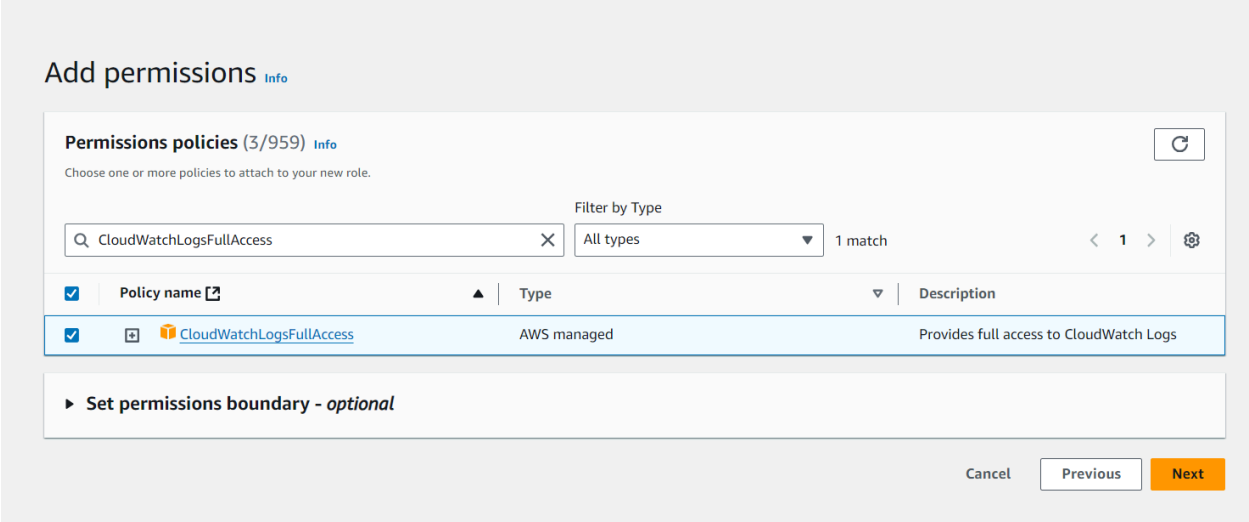
- AmazonS3ReadOnlyAccess(to allow read access to S3)

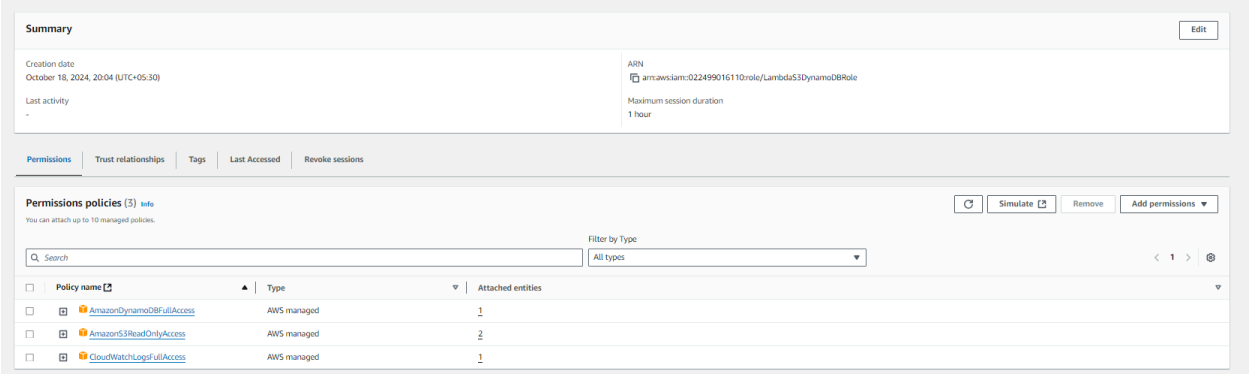


- AmazonDynamoDBFullAccess (or create a custom policy with specific permissions to write to DynamoDB).

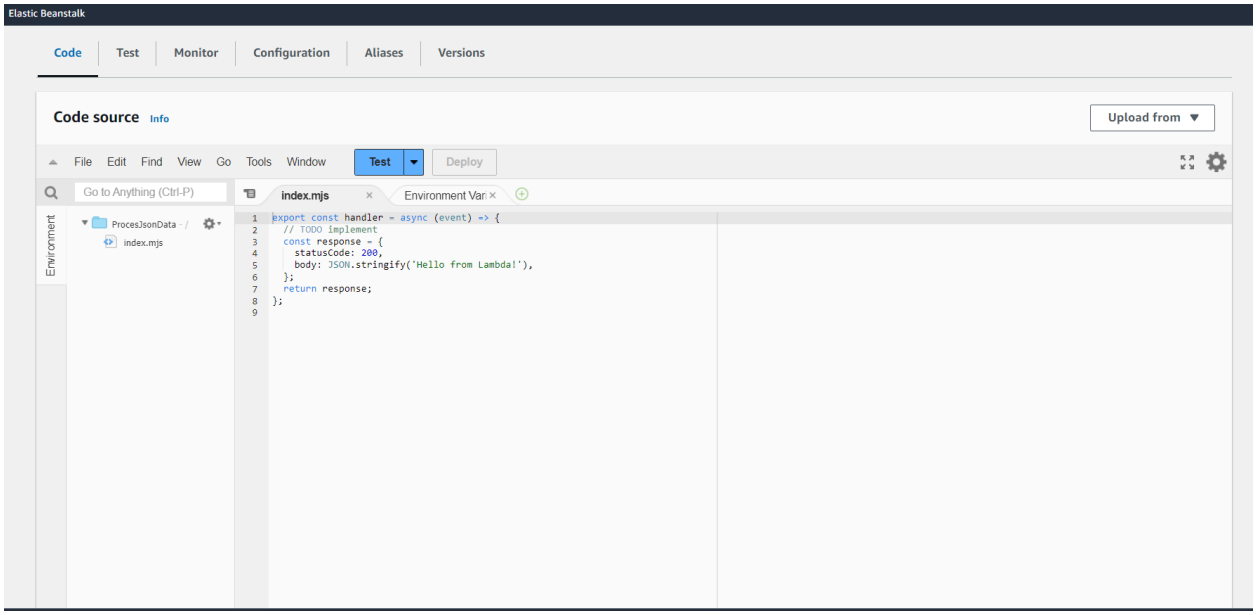


CloudWatchLogsFullAccess (permission to log function execution and errors.)





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).



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:

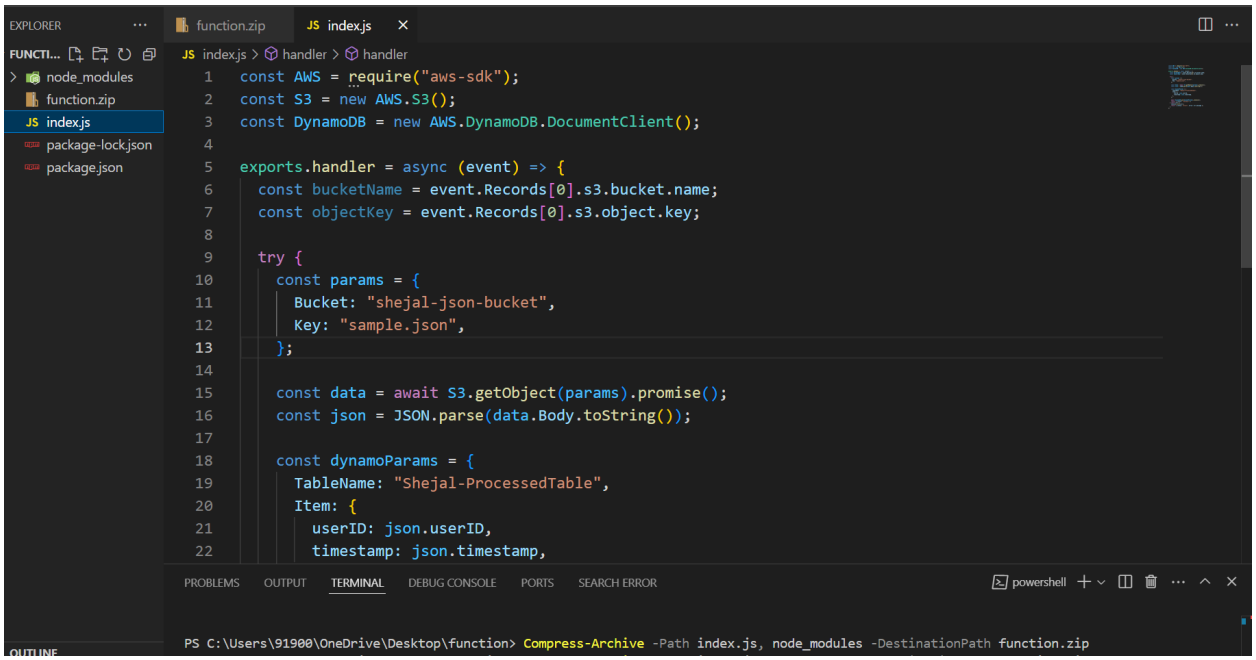
```bash
npm install aws-sdk
```

3. Create the Zip File:

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

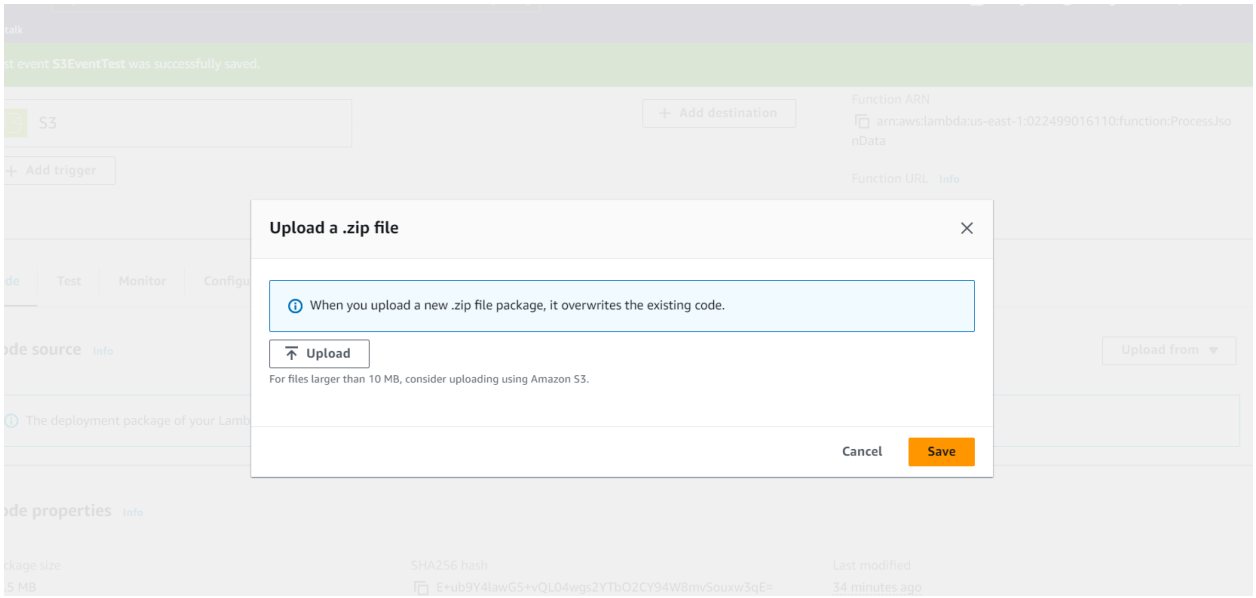
```bash
zip -r Function.zip .
```

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

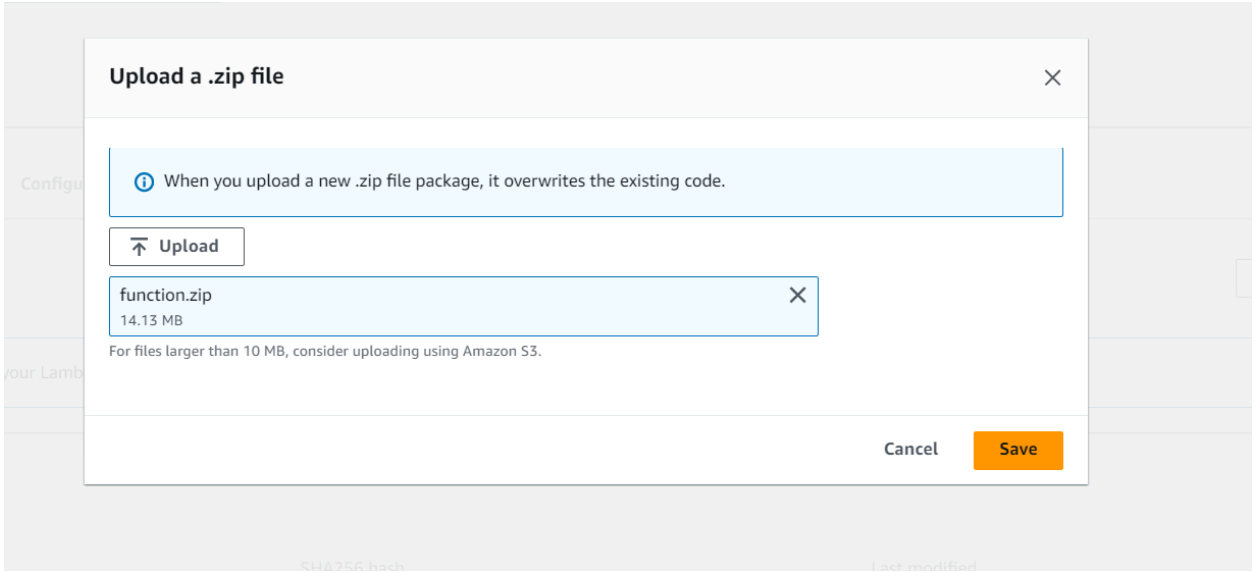


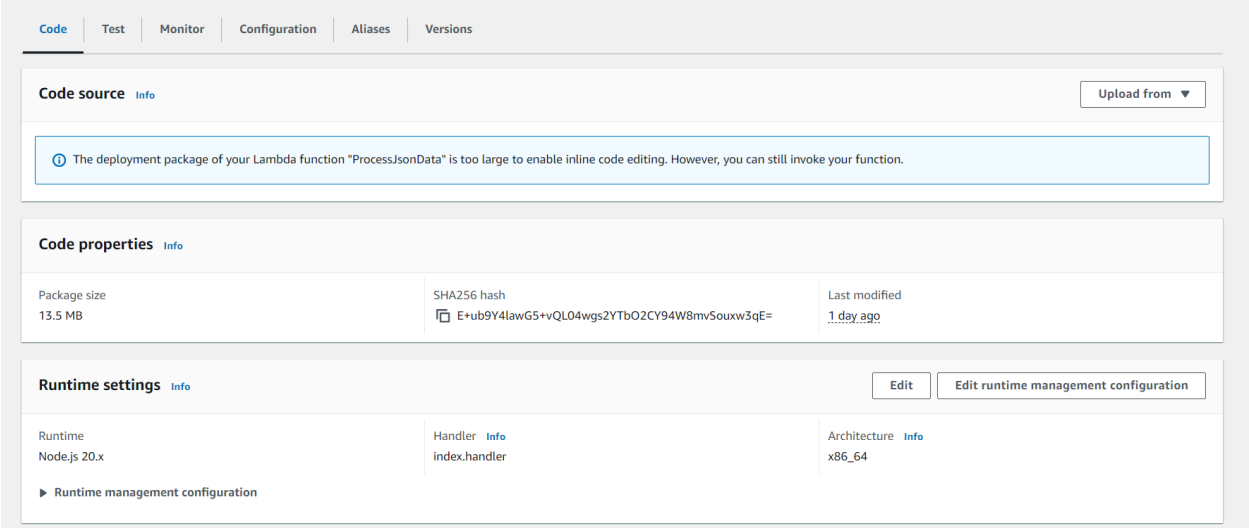
4. Upload the Zip File:

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



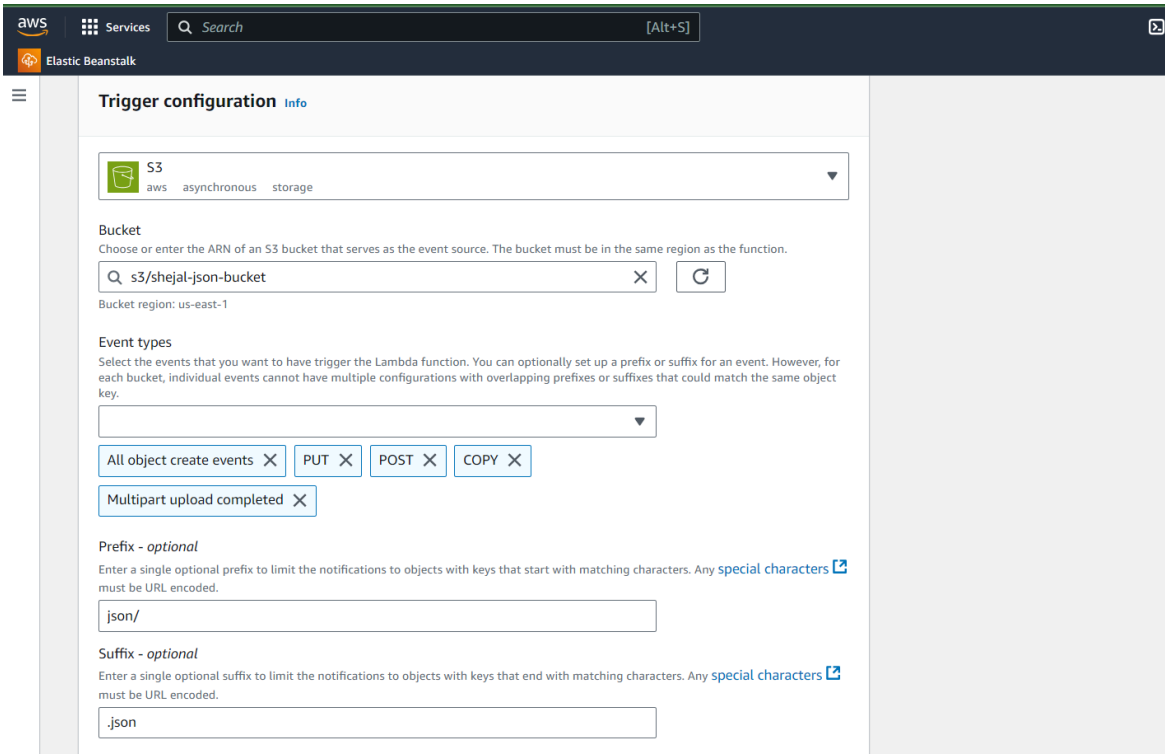
- Click "Upload," select your `MyLambdaFunction.zip`, and then click "Save."



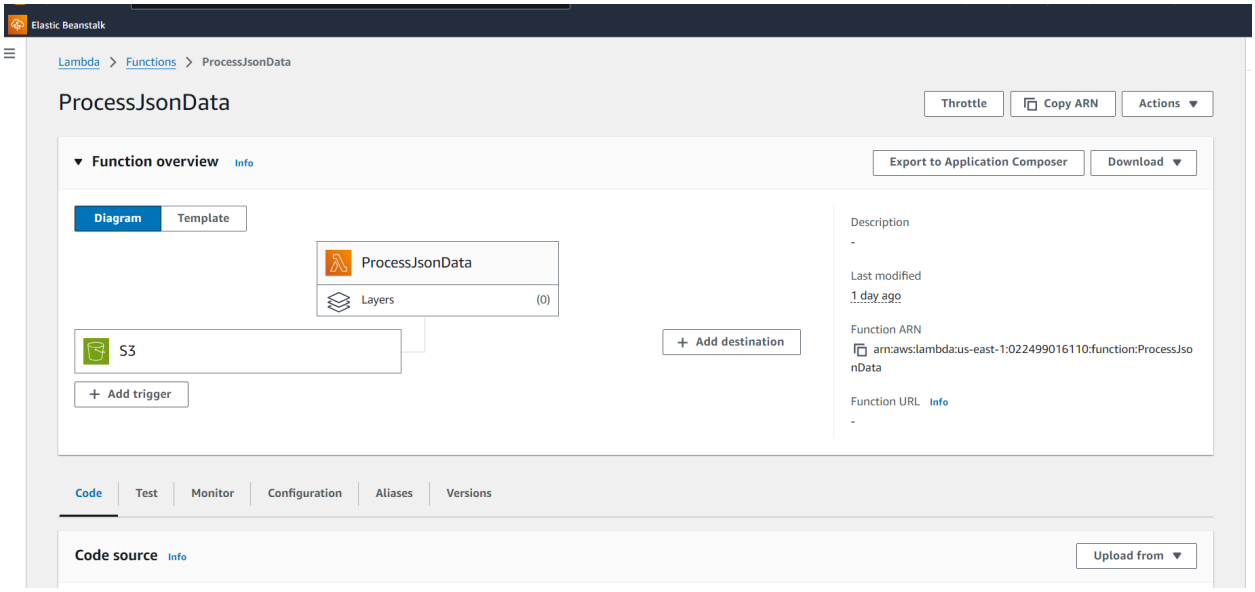


Step 3: Add an S3 Trigger to Your Lambda Function

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.



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

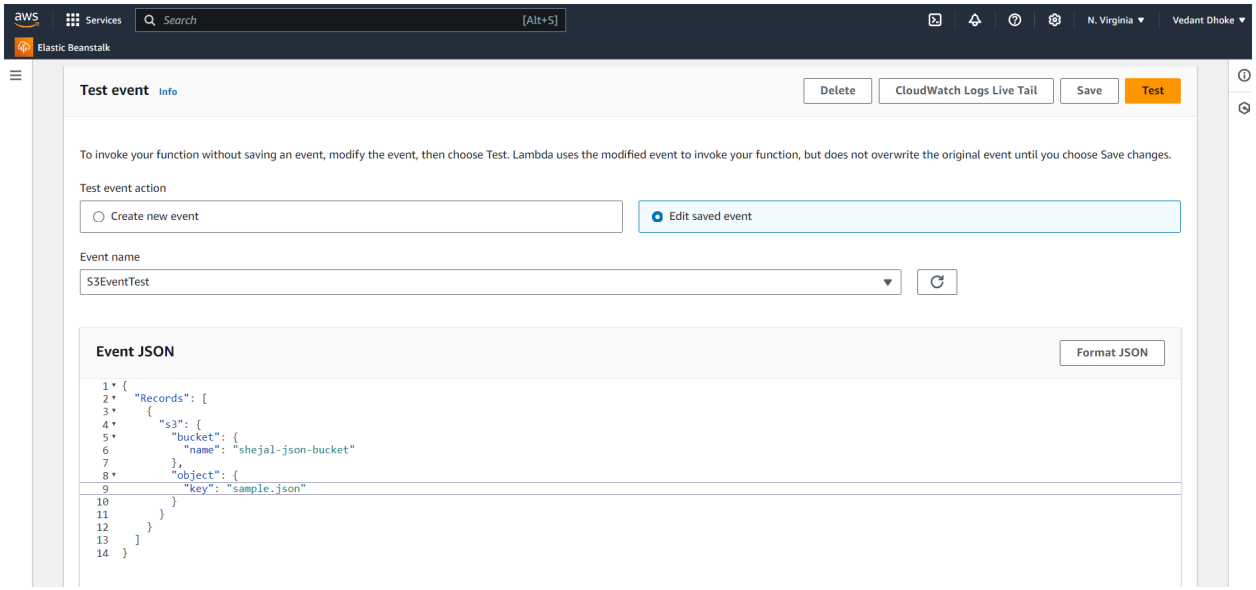
1. **Test the Function**:
    - Click on the **Test** tab within the Lambda function console.
    - If you haven't created a test event yet, you'll need to create one:
        - Click **Configure test event**.
        - Select **Create new test event**.

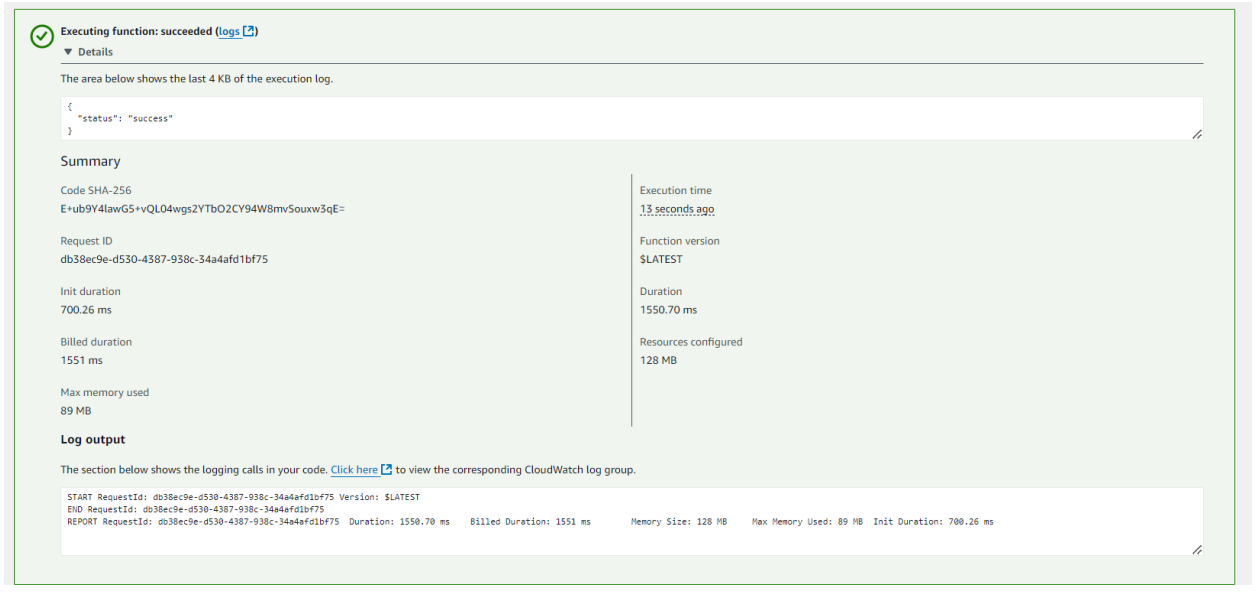
Name the event (e.g., TestEvent) and use the following JSON structure for the test event:
json
Copy code

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

- ■
- ■ Click **Create** to save the test event.

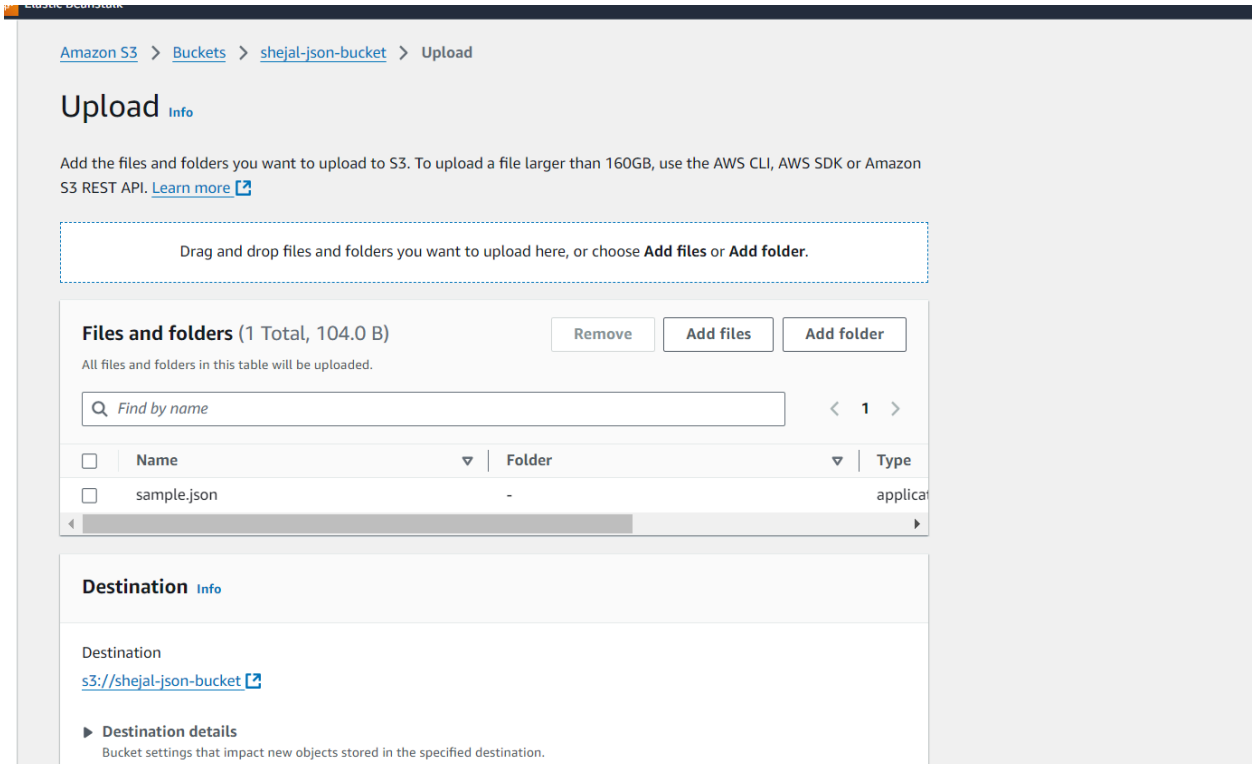
- Now, click the **Test** button to run the function with the test event. Check the **Execution result** and **Logs** to confirm the function executed without errors.





Step 5: Upload a Sample JSON File to S3

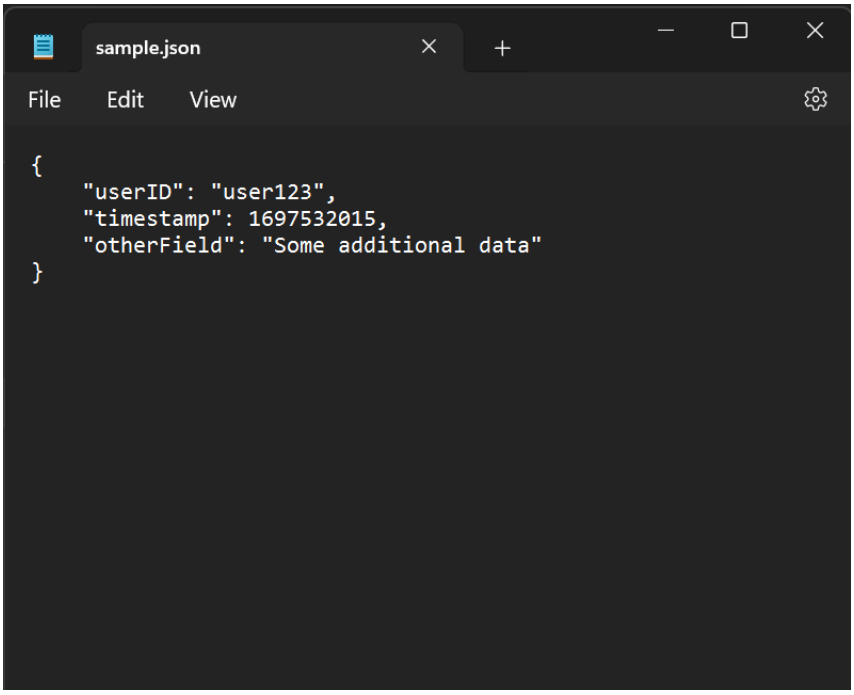
1. **Navigate to Your S3 Bucket**:
    - In the AWS Management Console, type "S3" in the search bar and select **S3** from the services list.
    - Click on your bucket named shejal-json-bucket to open it.



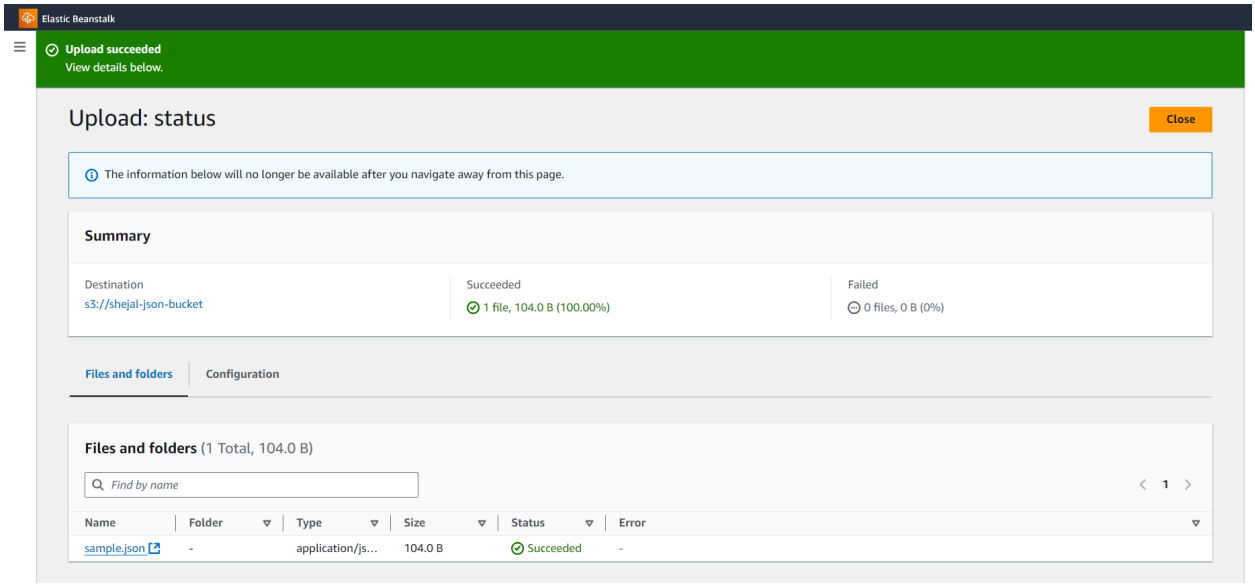
2. **Upload a Sample JSON File**:
    - Click on the **Upload** button to begin the upload process.

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"
}
```

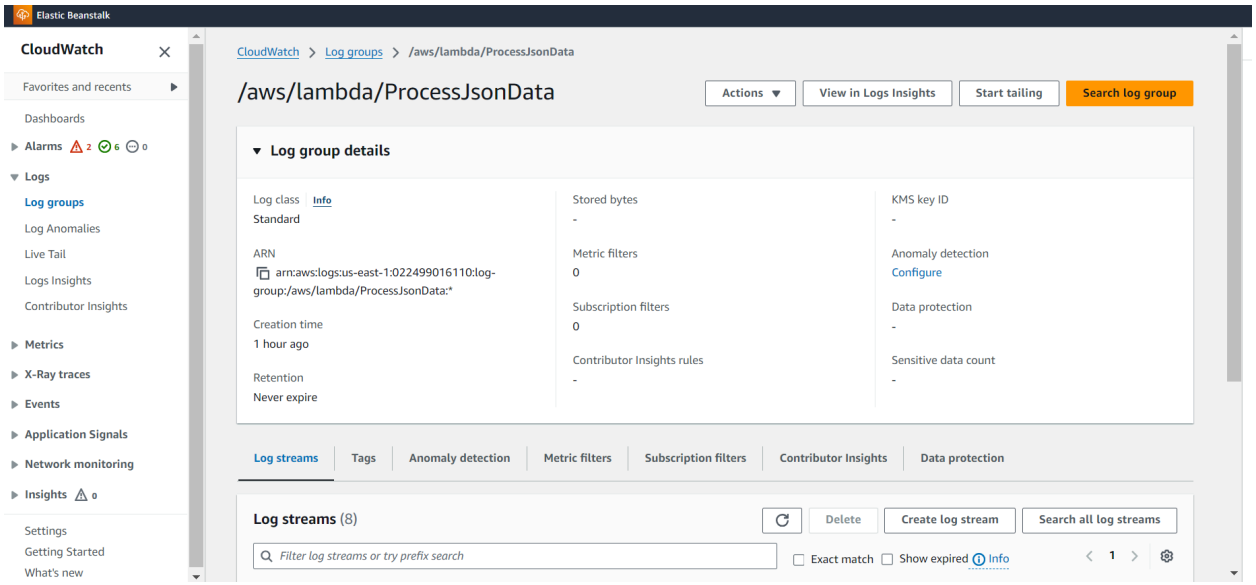


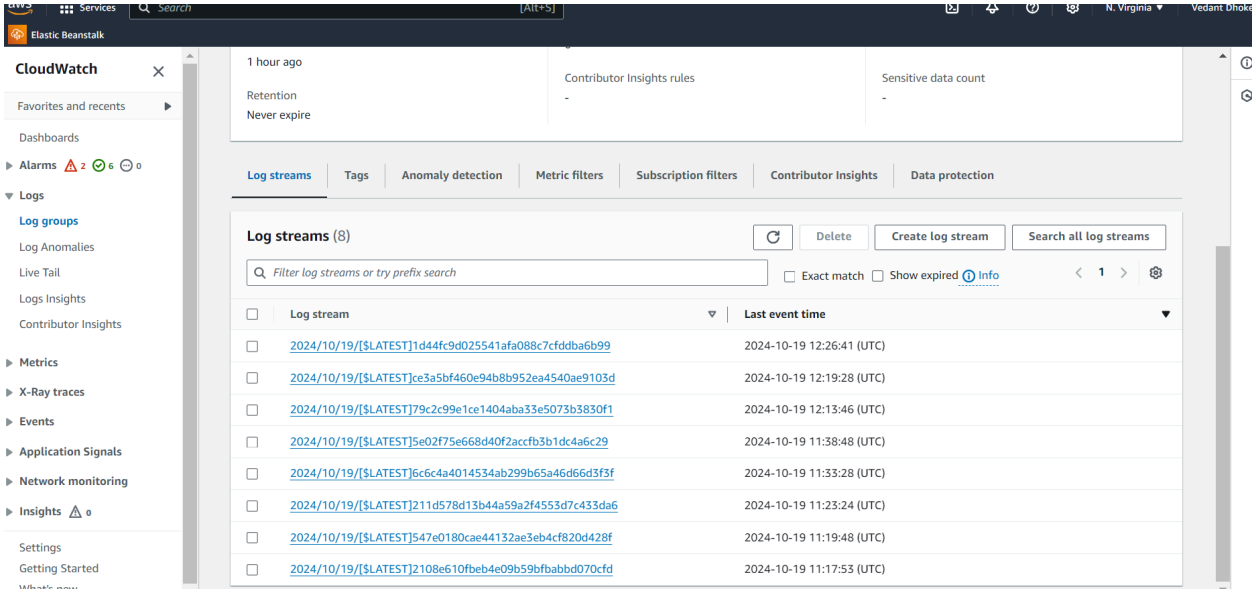
- After selecting the file, click **Upload** at the bottom of the dialog.

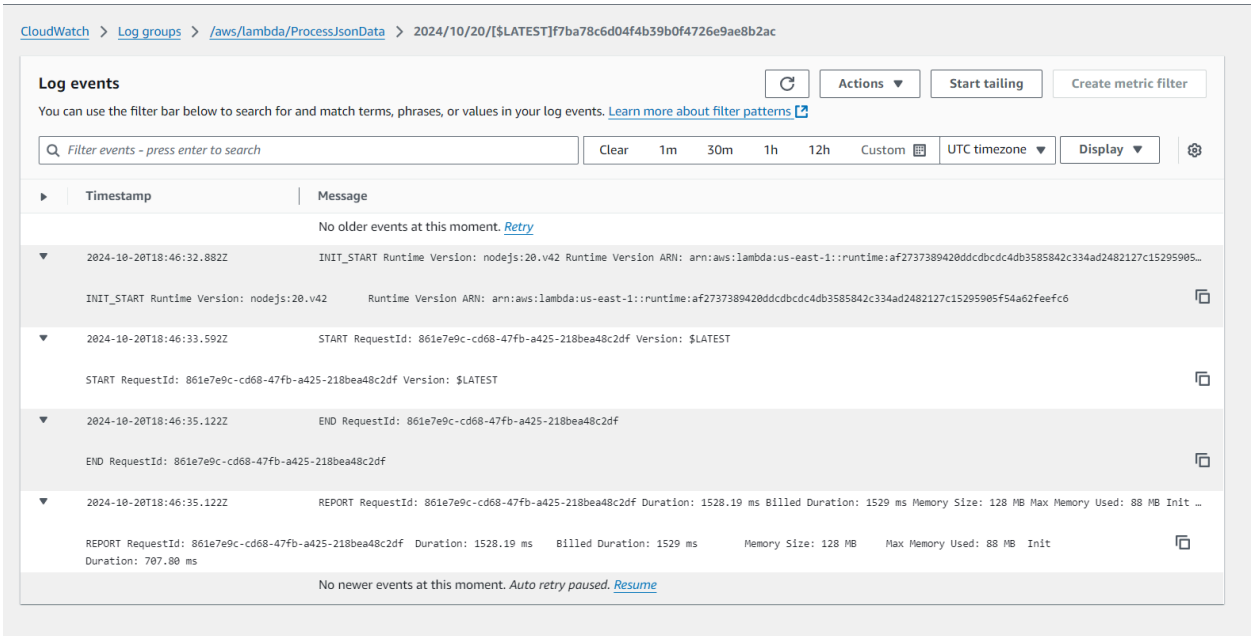


3. **Verify the Upload**:
   - Once the upload is complete, you should see `sample.json` listed in your S3 bucket contents.
   - You can click on the file to view its properties and confirm it was uploaded successfully.

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.

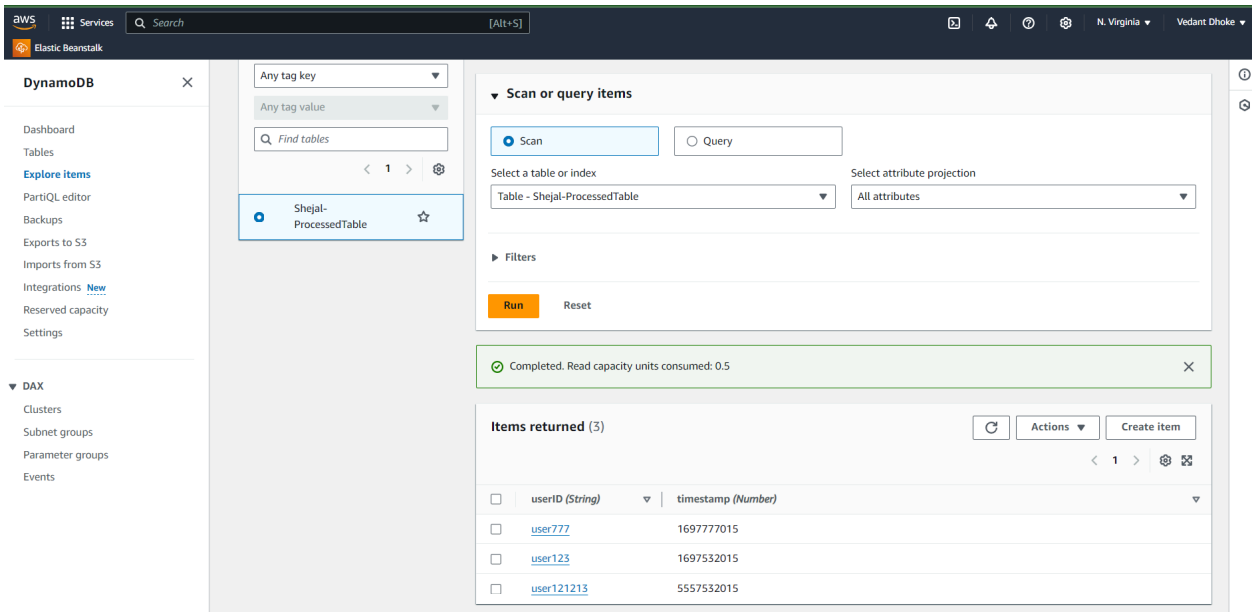






Step 6: Verify Data in DynamoDB

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.