Maniya Motiramani **D15B-38**

ADVANCED DEVOPS
CASE STUDY

I. Introduction:

In this case study on Real-Time Log Processing, the focus is on utilizing AWS services, specifically Lambda, CloudWatch, and S3, to address a log management challenge. The main objective was to establish an AWS Lambda function that triggers whenever a new log entry is added to a designated CloudWatch Log Group. This Lambda function, written in Python, filters log events based on a specified keyword, such as 'ERROR', and subsequently stores these filtered logs in an S3 bucket for further analysis and storage. This setup ensures efficient log management while providing real-time alerting and storage solutions, leveraging the seamless integration of AWS services.

Concepts Used: AWS Lambda, CloudWatch, S3.

AWS Lambda

AWS Lambda is a serverless computing service that allows you to run code without provisioning or managing servers. It automatically scales your applications by running code in response to triggers such as changes in data, updates to databases, or HTTP requests. With Lambda, you can focus on writing your application logic without worrying about the underlying infrastructure. It's highly cost-effective since you only pay for the compute time you consume. Lambda functions can be written in various programming languages, including Python, Java, and Node.js.

CloudWatch Log Group

CloudWatch Logs is part of Amazon CloudWatch, which provides monitoring and observability for AWS resources and applications. A Log Group in CloudWatch is a collection of log streams that share the same settings, such as retention, monitoring, and access control. Log streams are sequences of log events that share the same source, for example, log entries from a specific application or service. By using log groups, you can organize and manage your logs more effectively, set retention policies, and configure alarms to notify you of specific events.

Amazon S3

Amazon S3 (Simple Storage Service) is a scalable object storage service designed for a wide range of use cases, including data storage, backup and restore, archiving, and big data analytics. S3 provides a secure and highly available environment to store any amount of data from anywhere. You organize your data in buckets, which can hold an unlimited number of objects. S3 supports features like versioning, lifecycle policies, and cross-region replication to ensure data durability and availability. It integrates seamlessly with other AWS services, making it a cornerstone for many cloud-native applications

Key Feature and Application: Real-Time Log Processing

Unique Feature:

The ability to process logs in real-time, filtering for specific events (e.g., ERROR messages) as they are generated, and storing the relevant log entries in an S3 bucket.

Practical Use:

This feature enables quick detection and response to critical system events, such as errors or security breaches. By filtering and storing error logs in real-time, DevOps teams can efficiently track, manage, and address issues before they escalate, without manually sifting through large volumes of logs. This automated workflow streamlines troubleshooting and auditing processes, improving system reliability and uptime.

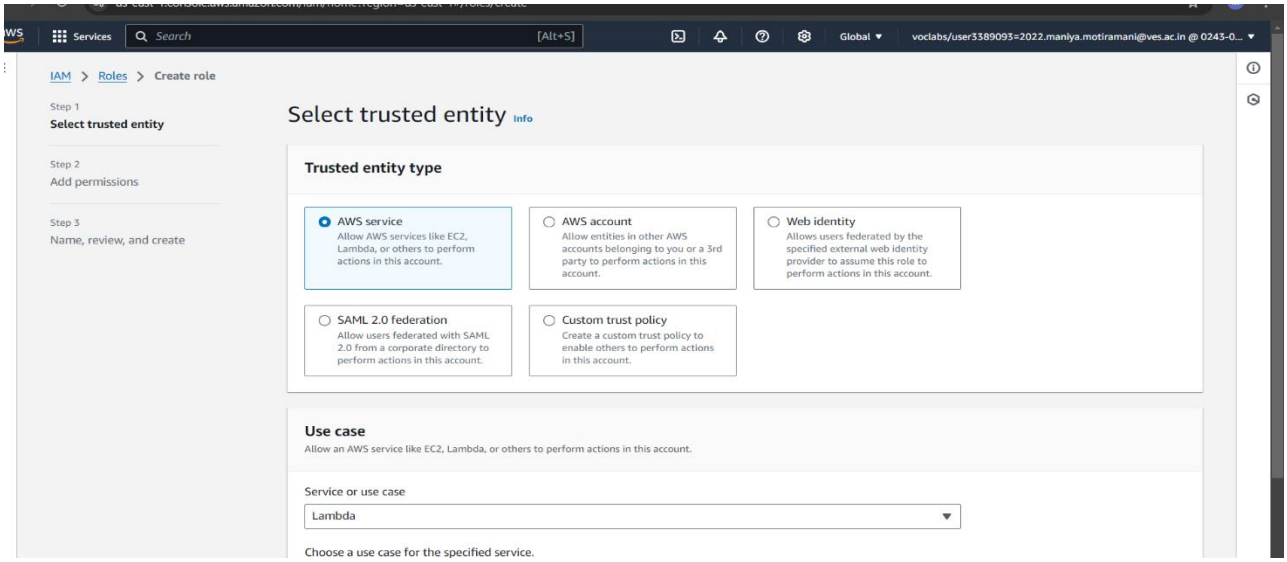
II. EXPLANATION

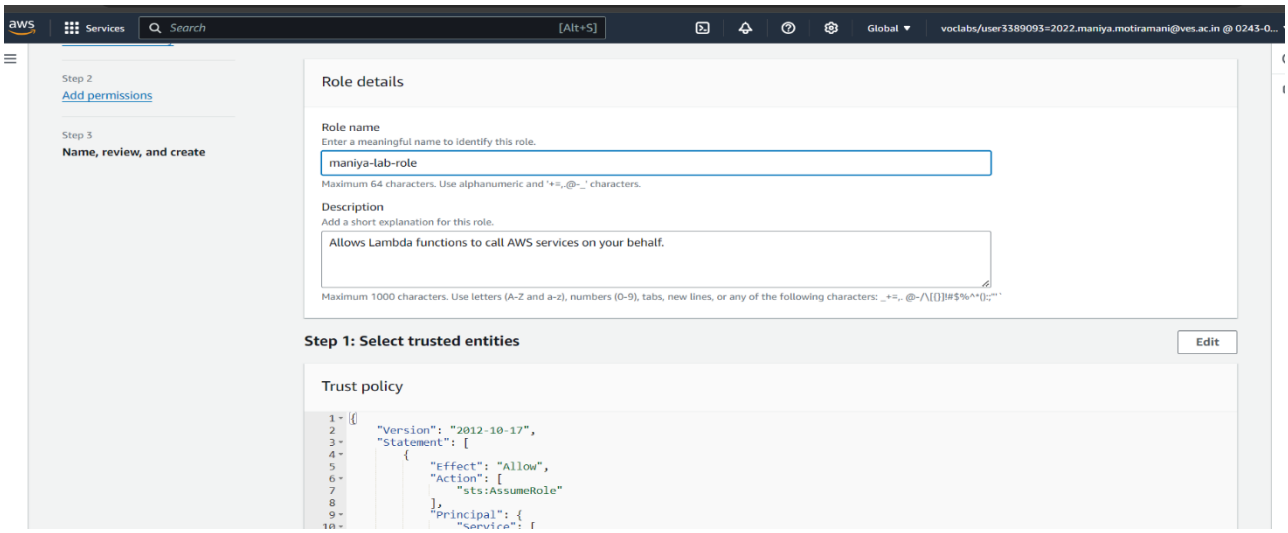
Step 1:Create a new IAM role

The first step in Real time log processing is to create an IAM role with necessary permissions

This user is essential for securely managing access to AWS services required for the project. By creating a dedicated IAM user, we can assign specific roles and permissions, ensuring that the Lambda function, CloudWatch Logs, and S3 bucket have the appropriate access and we don't face any permission related issues later on.

Add use case as Lambda
Add these policies to the permissions
AmazonS3FullAccess
AWSLambdaBasicExecutionRole
CloudWatchLogsFullAccess


Services
Search
[Alt+S]
Global
voclabs/user3389093=2022.maniya.motiramani@ves.ac.in @ 0243-0...
IAM > Roles > Create role
Step 1
Select trusted entity
Step 2
Add permissions
Step 3
Name, review, and create
Select trusted entity Info
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
Lambda
Choose a use case for the specified service.


Services
Search
[Alt+S]
Global
voclabs/user3389093=2022.maniya.motiramani@ves.ac.in @ 0243-0...
Step 2
Add permissions
Step 3
Name, review, and create
Role details
Role name
Enter a meaningful name to identify this role.
maniya-lab-role
Maximum 64 characters. Use alphanumeric and '+=,.@-_' characters.
Description
Add a short explanation for this role.
Allows Lambda functions to call AWS services on your behalf.
Maximum 1000 characters. Use letters (A-Z and a-z), numbers (0-9), tabs, new lines, or any of the following characters: _+=,. @-/\[{}]!#$%^*();:"'`
Step 1: Select trusted entities
Edit
Trust policy
{
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Action": [
"sts:AssumeRole"
],
"Principal": {

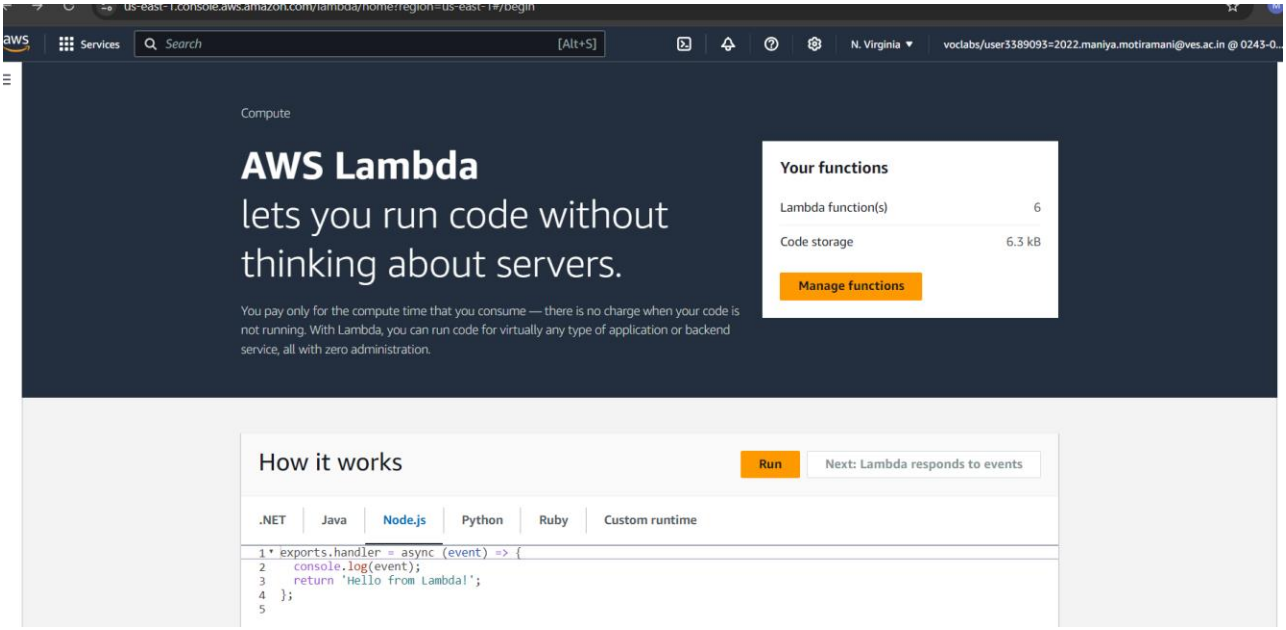
Step 2: Add permissions Edit

Permissions policy summary

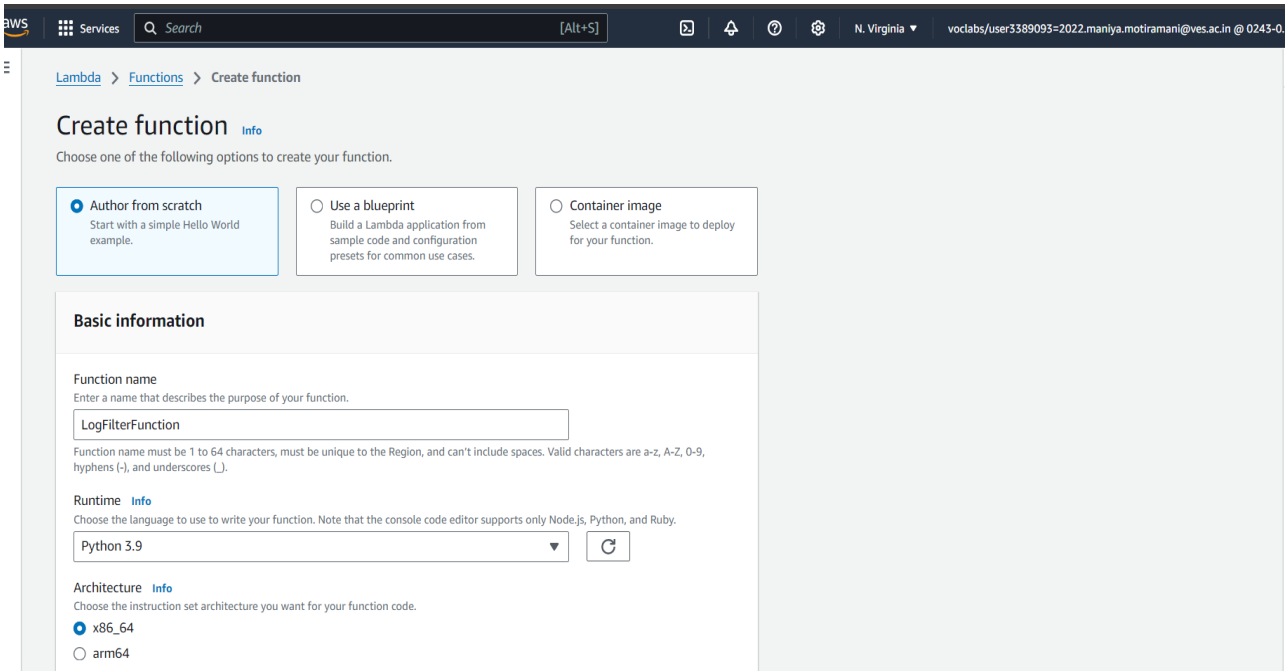
Policy name	Type	Attached as
AmazonS3FullAccess	AWS managed	Permissions policy
AWSLambdaBasicExecutionRole	AWS managed	Permissions policy
CloudWatchLogsFullAccess	AWS managed	Permissions policy

Step 3: Add tags

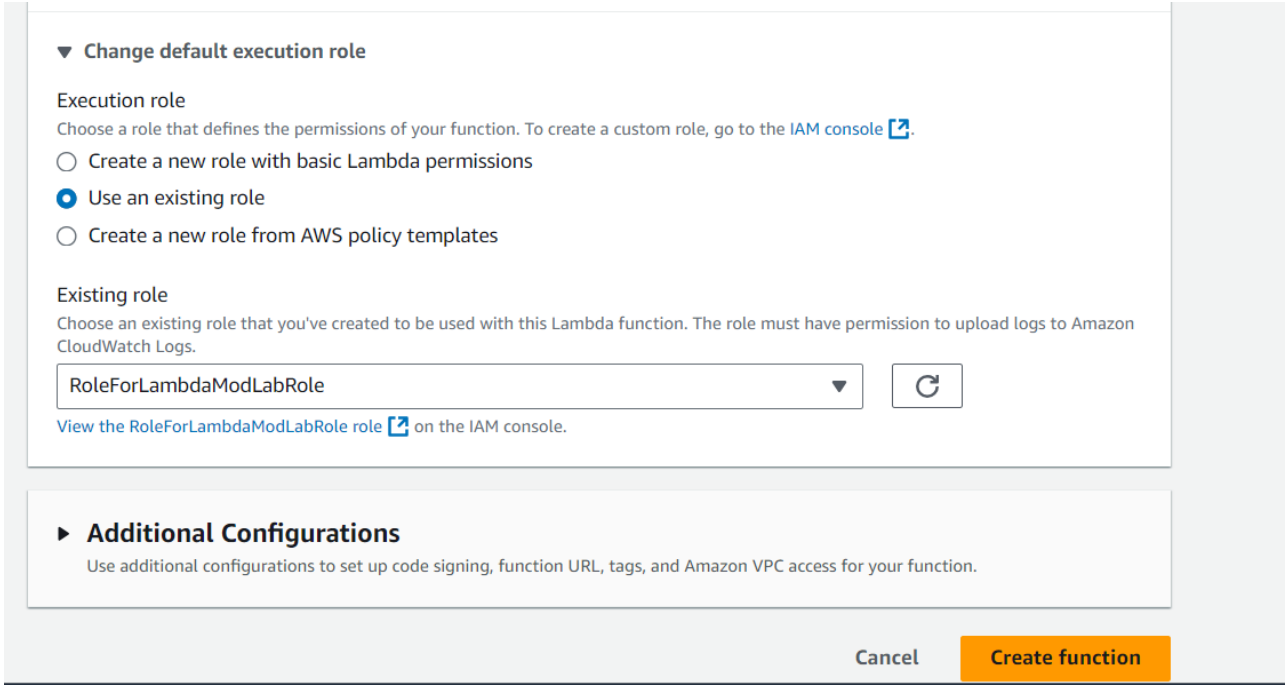
Step 2:Create AWS lambda function



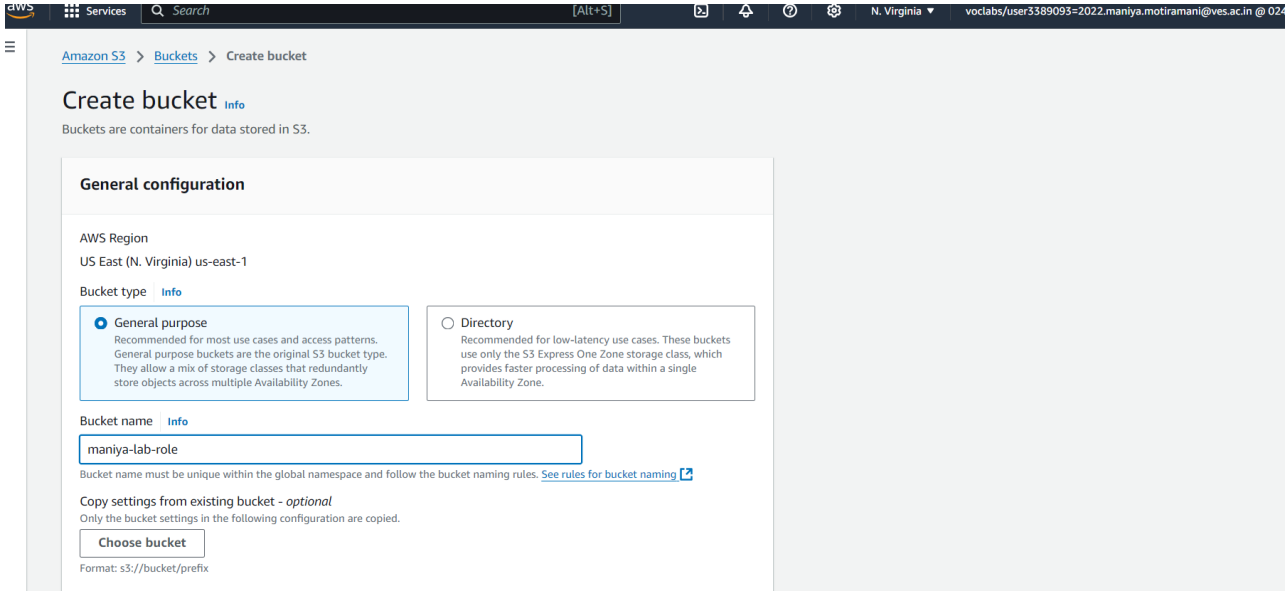
Give your function a name and choose python as your runtime language

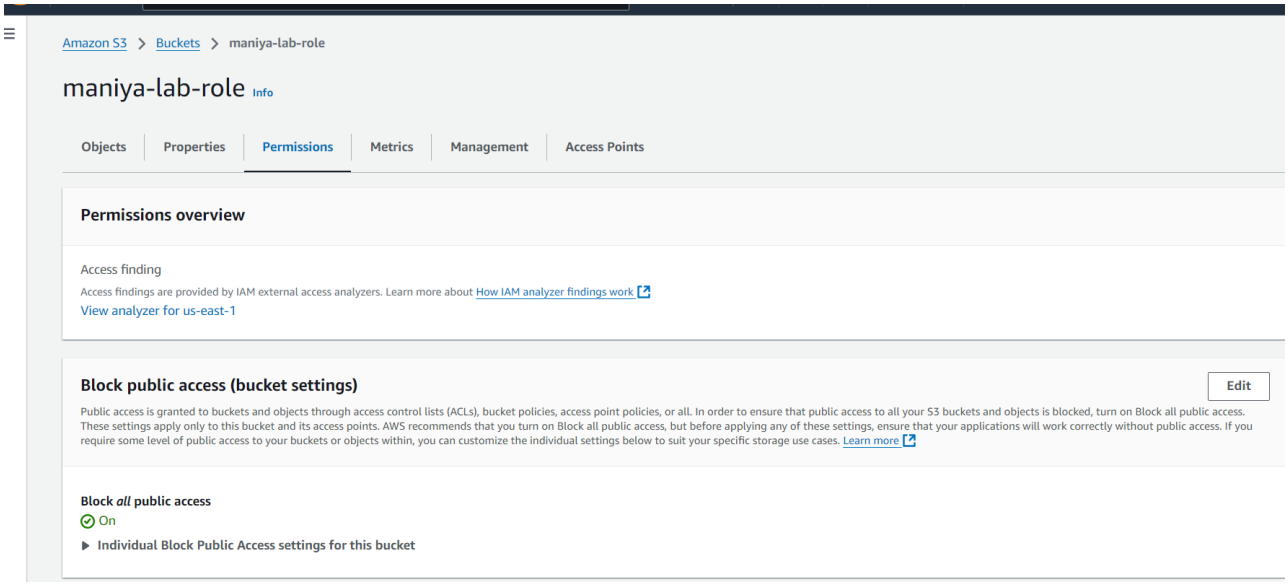


Add role created in Step 1



Step 3:Create S3 bucket





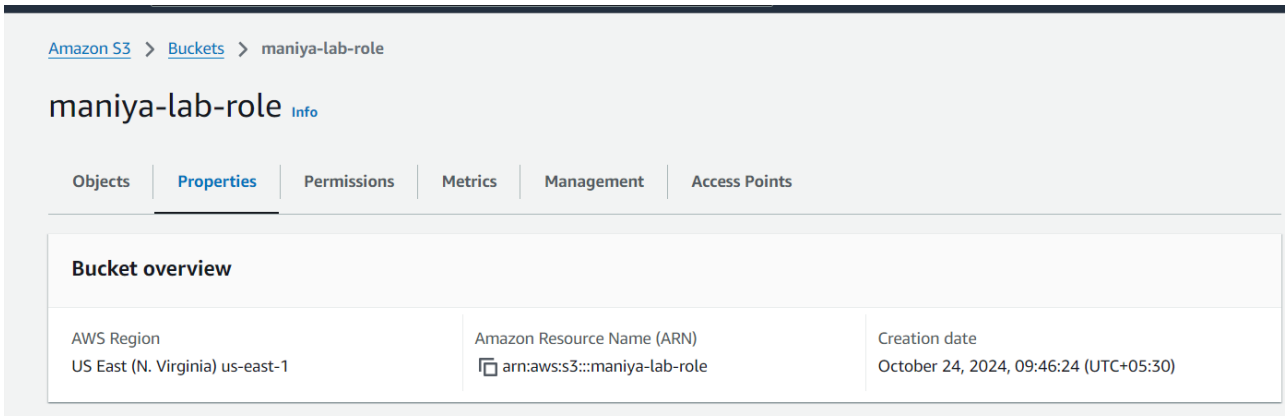
Edit the bucket policy

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Principal": {
            "AWS": "arn:aws:iam::724772084448:role/LambdaLogProcessorRole"
         },
         "Action": "s3:PutObject",
         "Resource": "arn:aws:s3:::maniya-logs-bucket/*"
      }
   ]
}
```

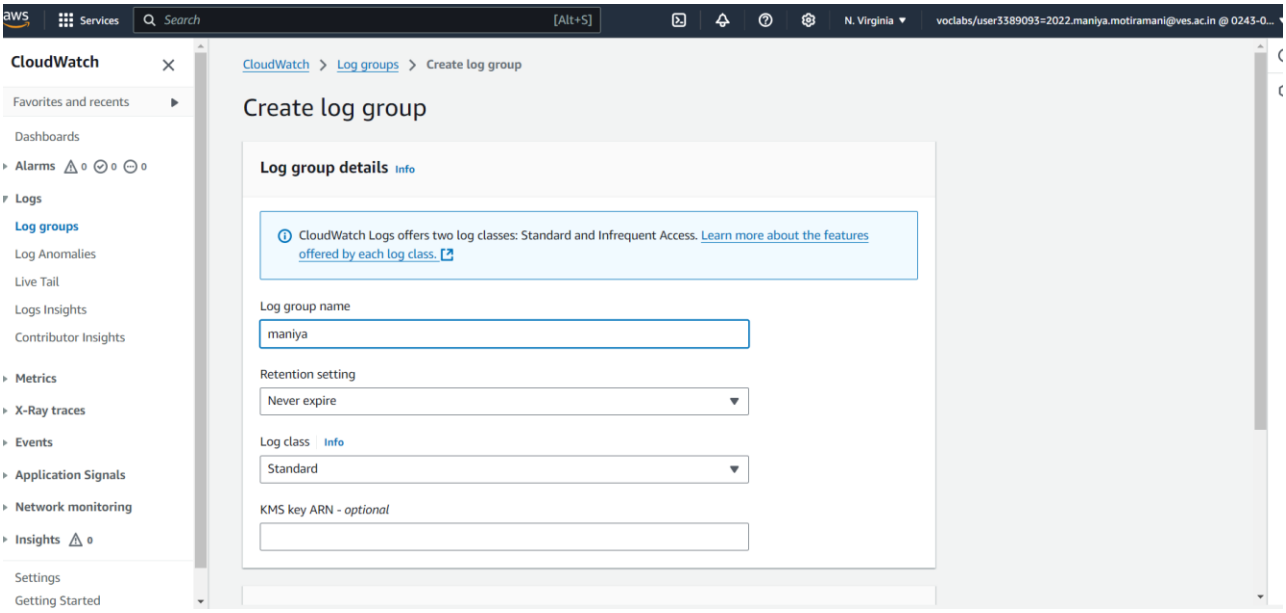
Here, 724772084448 is my account id for AWS and LambdaLogProcessorRole is the IAM role created in step 1.

"Resource": "arn:aws:s3:::maniya-lab-role

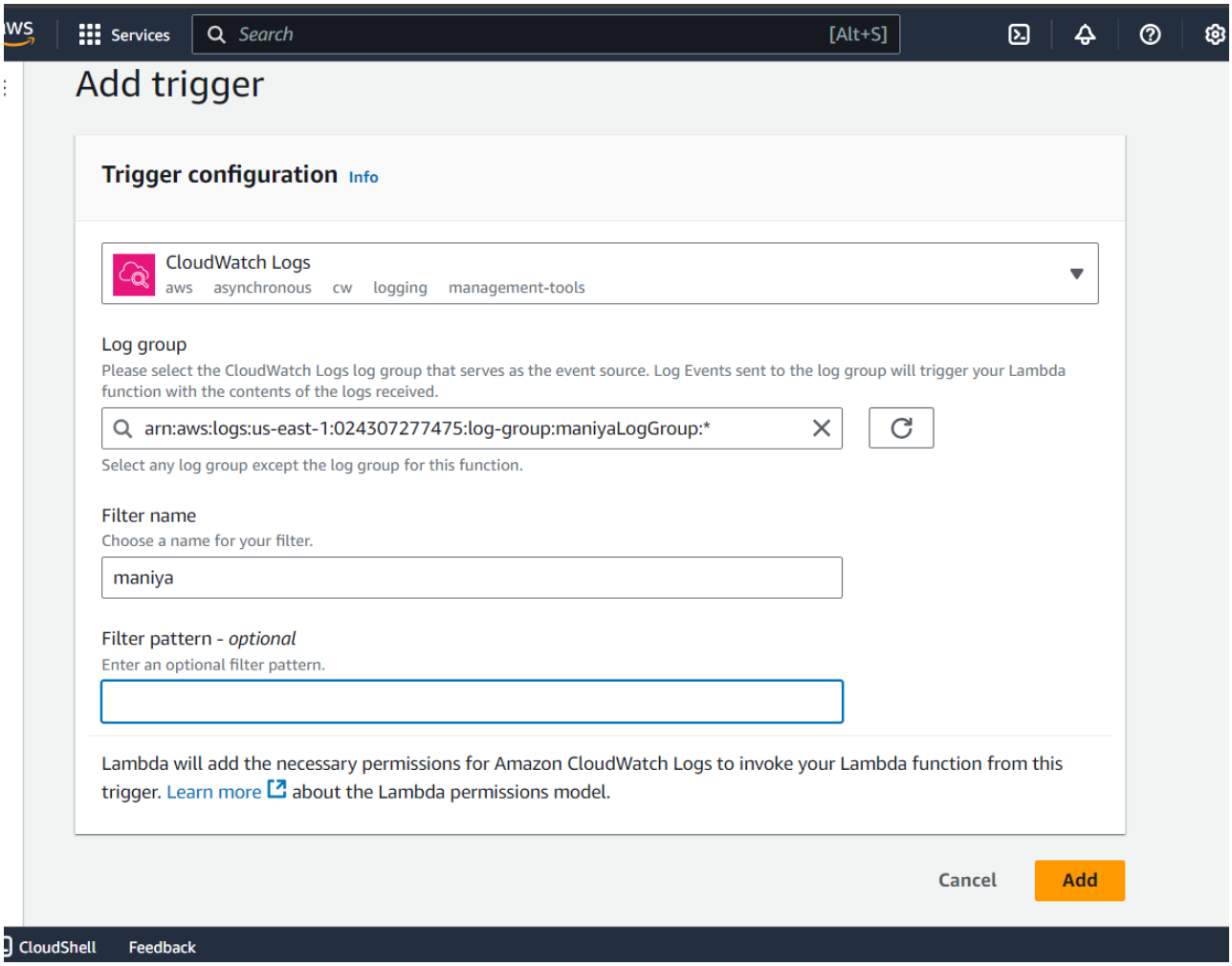
/ : this specifies the name of my S3 bucket



Step 4: Create CloudWatch groups



Step 5:Add the cloudwatch group trigger in Lambda function

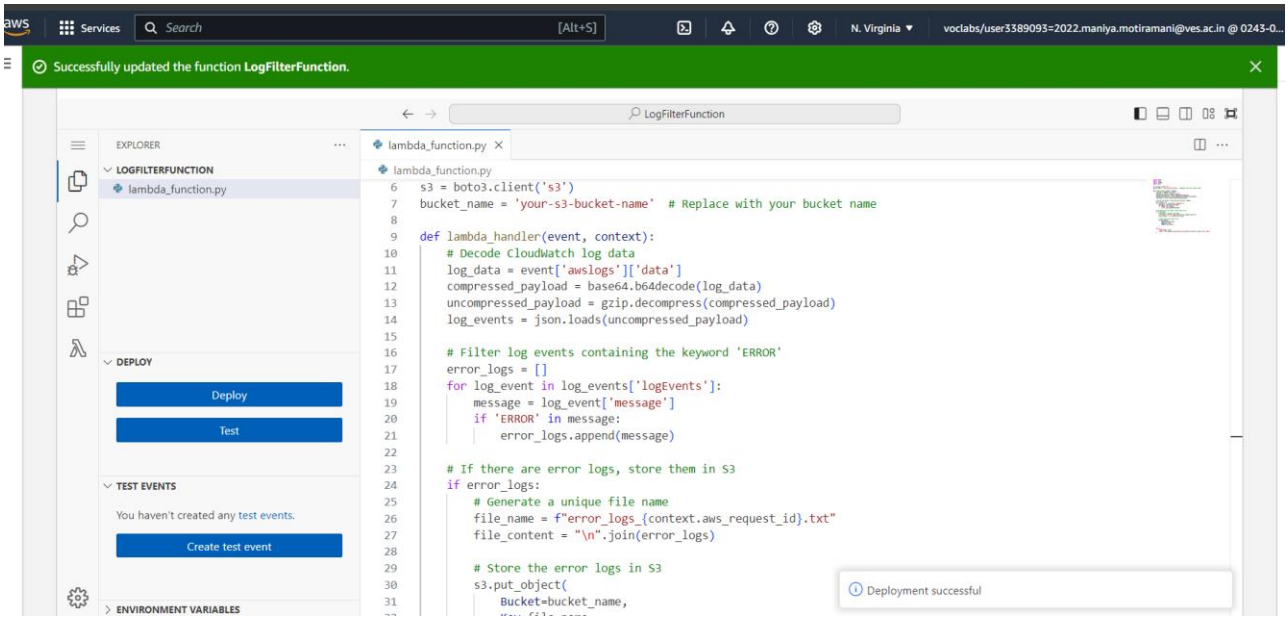


Step 6: In the lambda function add the code:

```
import boto3
import json
import time

s3_client = boto3.client('s3')

def lambda_handler(event, context):
    try:
        # Debug: Print the full event to understand its structure
        print("Event Received: ", json.dumps(event, indent=2))

        log_events = event['logEvents'] # Extract log events
        print(f"Received {len(log_events)} log events")

        # Filter logs containing 'ERROR'
        filtered_logs = [log for log in log_events if 'ERROR' in log['message']]
        print(f"Filtered {len(filtered_logs)} error log events")

        if filtered_logs:
            # Generate a unique key for each log upload to avoid overwriting
            timestamp = int(time.time())
            s3_client.put_object(
                Bucket='shravani-logs-bucket',
                Key=f'filtered_logs_{timestamp}.json', # Unique key
                Body=json.dumps(filtered_logs)
            )
            print(f"Successfully uploaded filtered logs to S3 with key:
filtered_logs_{timestamp}.json")

        return {
            'statusCode': 200,
            'body': json.dumps('Logs processed successfully!')
        }
```

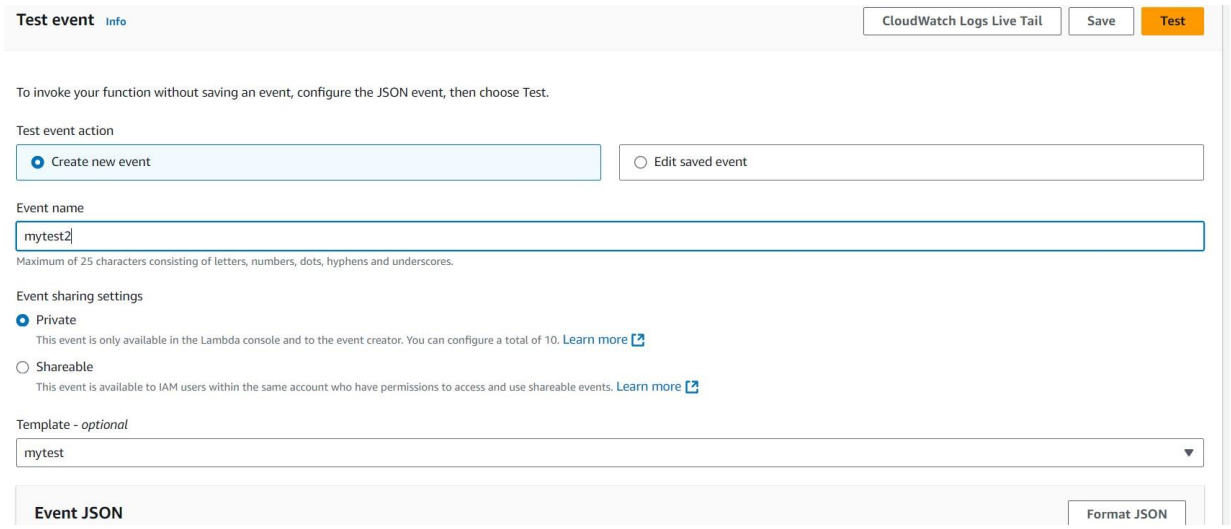
```
    except KeyError as e:
        print(f"KeyError: {e}")
        raise e
    except Exception as e:
        print(f"Exception: {e}")
        raise e
```

Note: In the above code, add the name of S3 bucket created earlier.



Deploy to save the code

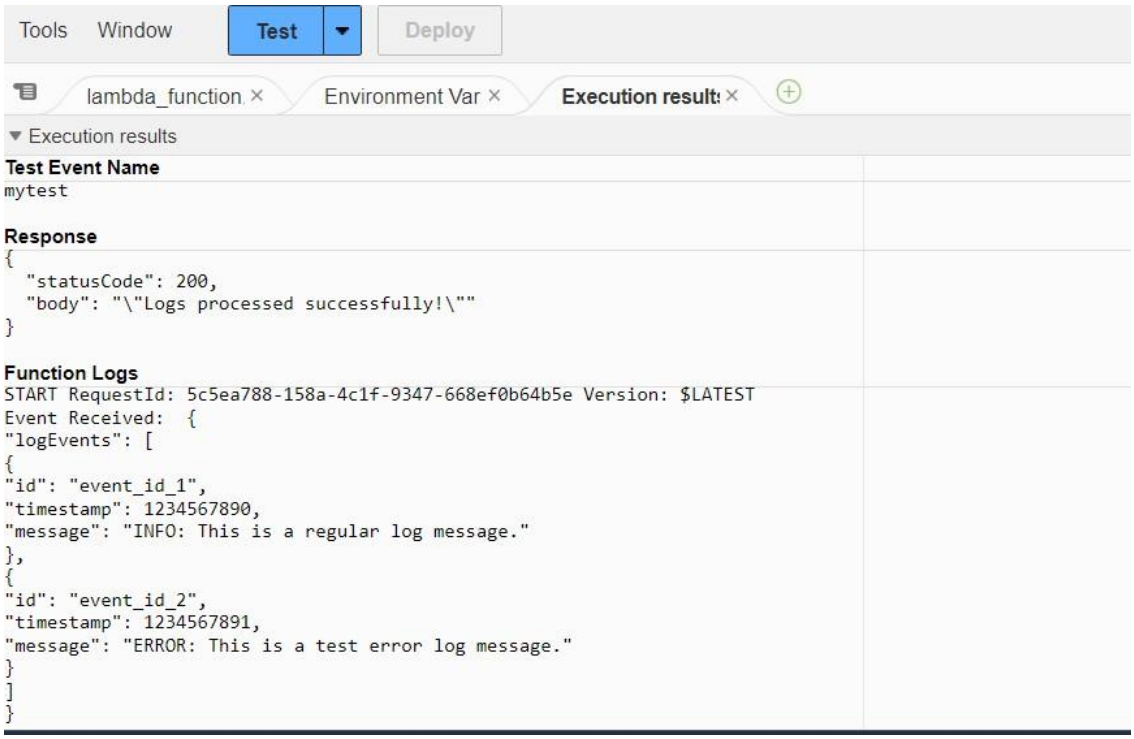
Step 7: Create a new test event to test the setup.



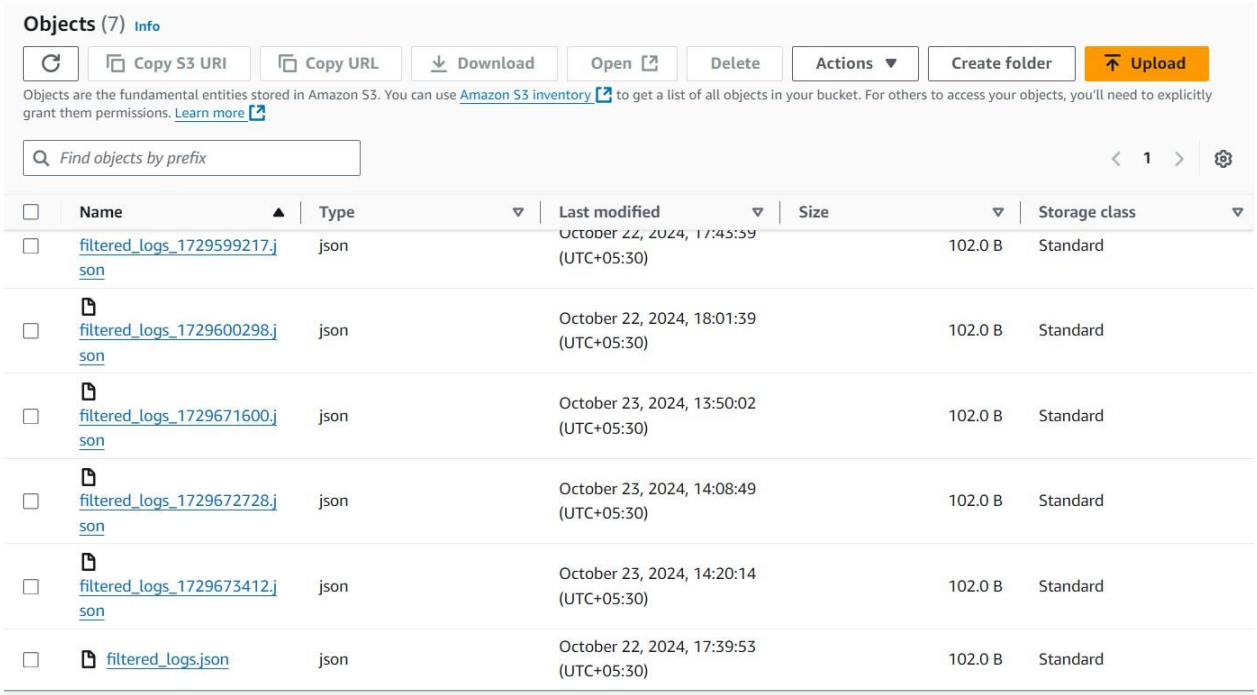
Add this to the json part

```
{
  "logEvents": [
    {
      "id": "event_id_1",
      "timestamp": 1234567890,
      "message": "INFO: This is a regular log message."
    },
    {
      "id": "event_id_2",
      "timestamp": 1234567891,
      "message": "ERROR: This is a test error log message."
    }
  ]
}
```

Step 8: Test the code



Step 9: To verify, go back to your S3 bucket. A new item called filtered_logs.json is added in the S3 bucket



On opening the json file, we can see the output.

Guidelines:

1. **Use a personal AWS account** if your AWS Academy account lacks sufficient privileges for the default role. This ensures you have the necessary access to all services without limitations.
2. **Principle of Least Privilege**: Assign only the minimum permissions required for IAM roles and policies. This reduces the potential security risks by limiting access to only essential AWS resources.
3. **Logging and Monitoring**: Enable comprehensive logging for your Lambda functions and monitor their performance using CloudWatch. This approach enhances your ability to troubleshoot issues quickly and ensures the system runs smoothly.

Conclusion:

In this case study, the integration of **AWS Lambda**, **CloudWatch Logs**, and **S3** for real-time log processing was successfully demonstrated. To ensure secure access to the necessary AWS services, an **IAM user** with fine-grained permissions was created, following the principle of least privilege. A **Lambda function** was configured to automatically trigger when new log entries were added to a **CloudWatch Log Group**, filtering specific log events based on predefined keywords, such as ERROR. The filtered logs were then stored in an **S3 bucket** for efficient analysis and long-term storage.

This setup significantly enhanced the system's log monitoring, alerting, and troubleshooting capabilities by automating the log management process. Following best practices, such as minimizing resource access and implementing robust error handling and retries within the Lambda function, ensured that the solution was secure, reliable, and scalable. This architecture effectively streamlines operational workflows, enabling real-time insights and improving overall system observability.