Mahvish Siddiqui
D15A 58

AdvDevops Case Study

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

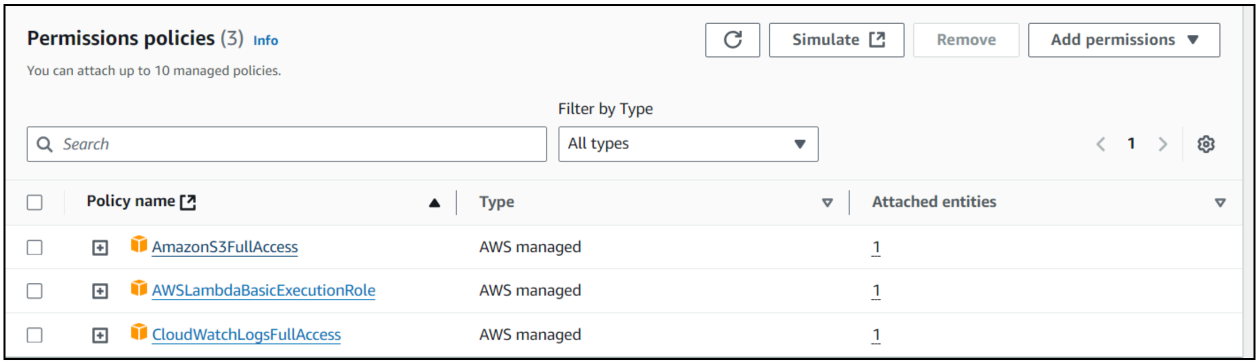
CloudWatch Logs is part of Amazon CloudWatch, which provides monitoring and observability for AWS resources and applications. A Log Group in CloudWatch is a collection of log streams that share the same settings, such as retention, monitoring, and access control. Log streams are sequences of log events 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.

Step-by-Step Explanation

Step 1: Create a new IAM role and add these policies
The first step in setting up the real-time log processing system was to create a new IAM user with the 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.

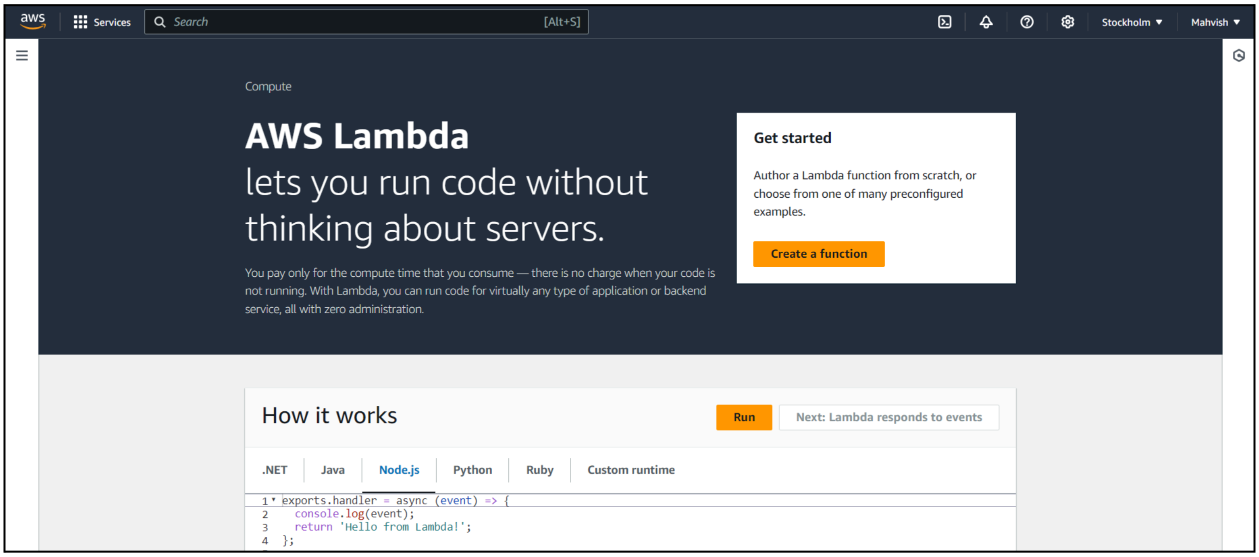


Permissions policies (3) Info

You can attach up to 10 managed policies.

□	Policy name	Type	Attached entities
□	AmazonS3FullAccess	AWS managed	1
□	AWSLambdaBasicExecutionRole	AWS managed	1
□	CloudWatchLogsFullAccess	AWS managed	1

Step 2: Go to AWS Lambda console and create a new function.



AWS Lambda
lets you run code without thinking about servers.

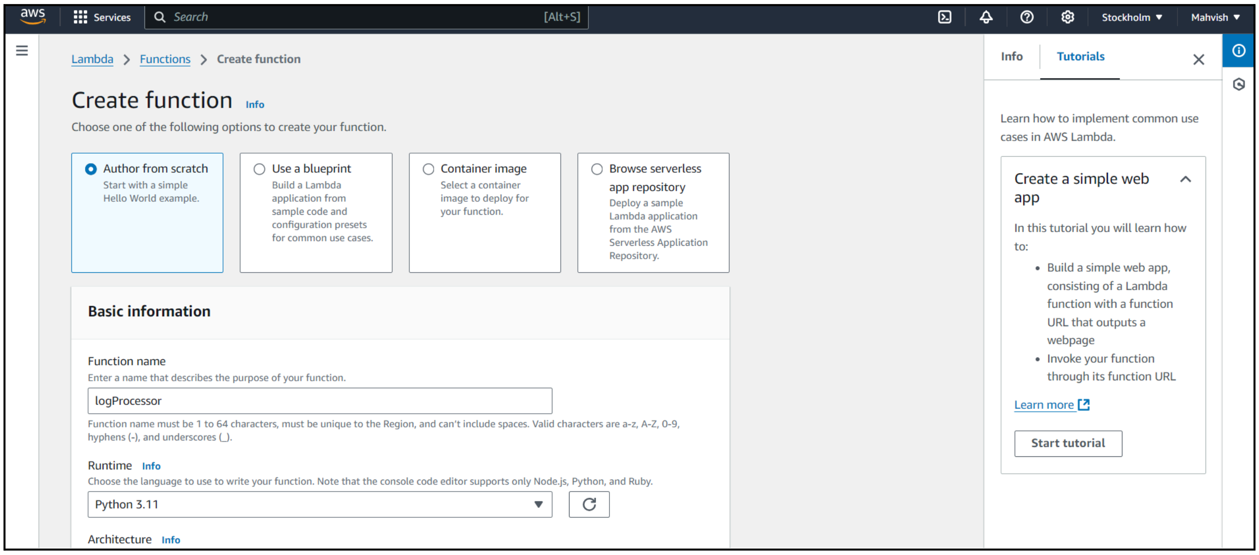
You pay only for the compute time that you consume — there is no charge when your code is not running. With Lambda, you can run code for virtually any type of application or backend service, all with zero administration.

Get started

Author a Lambda function from scratch, or choose from one of many preconfigured examples.

```
exports.handler = async (event) => {
  console.log(event);
  return 'Hello from Lambda!';
};
```

Step 3: Give your function a name and choose python as your runtime language



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.

○ Browse serverless app repository
Deploy a sample Lambda application from the AWS Serverless Application Repository.

Basic information

Function name
Enter a name that describes the purpose of your function.
logProcessor
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.
Python 3.11

Architecture Info

Learn how to implement common use cases in AWS Lambda.

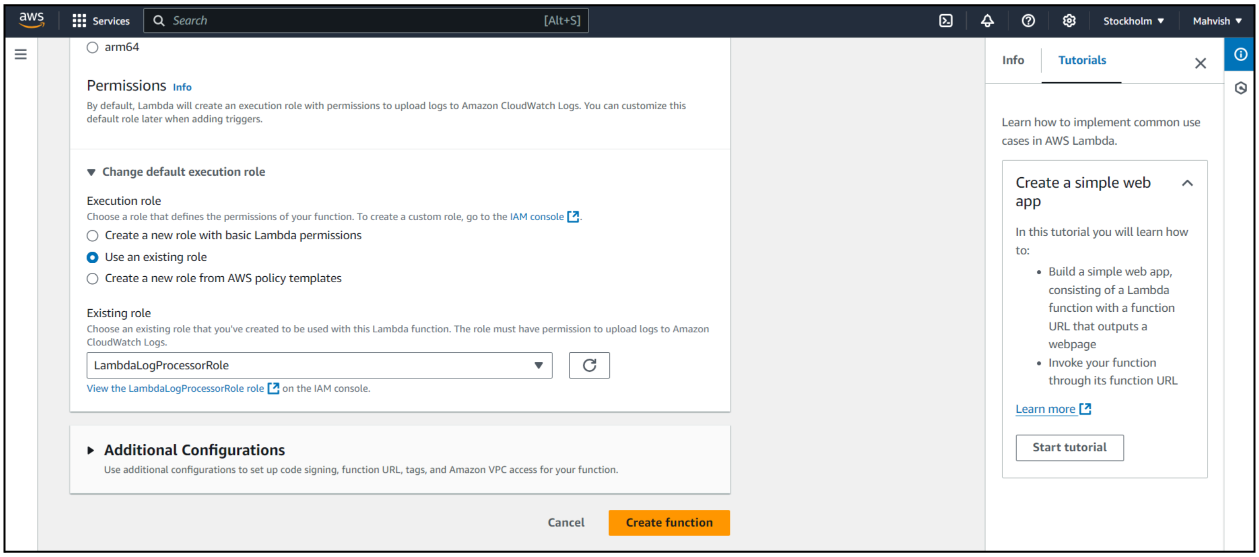
Create a simple web app

In this tutorial you will learn how to:

- Build a simple web app, consisting of a Lambda function with a function URL that outputs a webpage
- Invoke your function through its function URL

Learn more

Step 4: Add the IAM role created earlier to your lambda function.



○ arm64

Permissions Info

By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

▼ Change default execution role

Execution role
Choose a role that defines the permissions of your function. To create a custom role, go to the IAM console.

○ Create a new role with basic Lambda permissions
● Use an existing role
○ Create a new role from AWS policy templates

Existing role
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.
LambdaLogProcessorRole
View the LambdaLogProcessorRole role on the IAM console.

▶ Additional Configurations
Use additional configurations to set up code signing, function URL, tags, and Amazon VPC access for your function.

Learn how to implement common use cases in AWS Lambda.

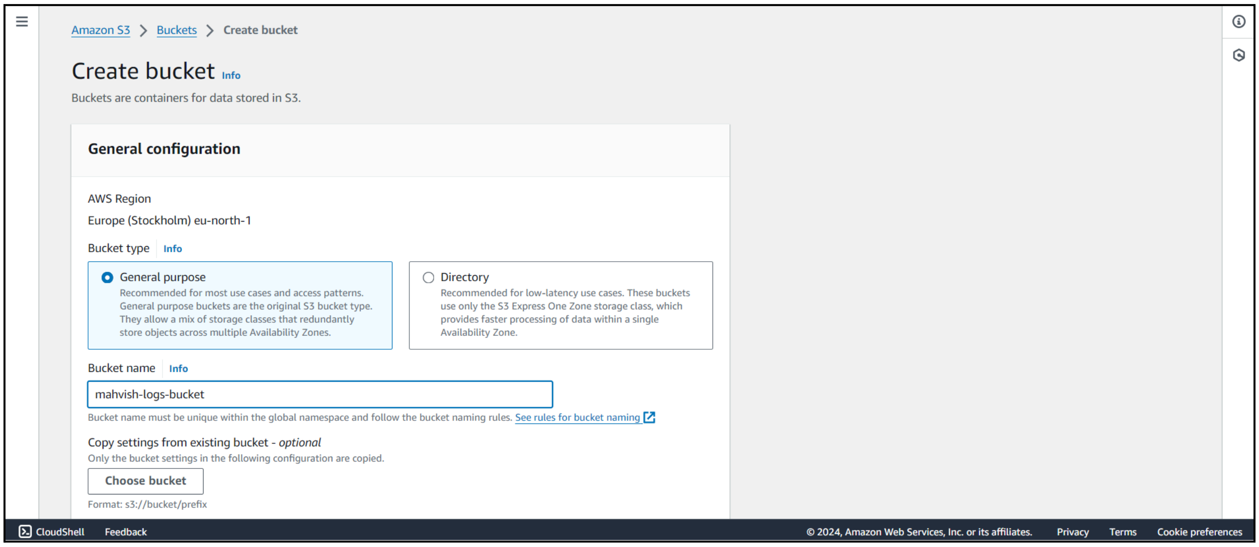
Create a simple web app

In this tutorial you will learn how to:

- Build a simple web app, consisting of a Lambda function with a function URL that outputs a webpage
- Invoke your function through its function URL

Learn more

Step 5: Create an S3 bucket. Keep the default settings.



Create bucket Info

Buckets are containers for data stored in S3.

General configuration

AWS Region

Europe (Stockholm) eu-north-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

mahvish-logs-bucket

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.

Choose bucket

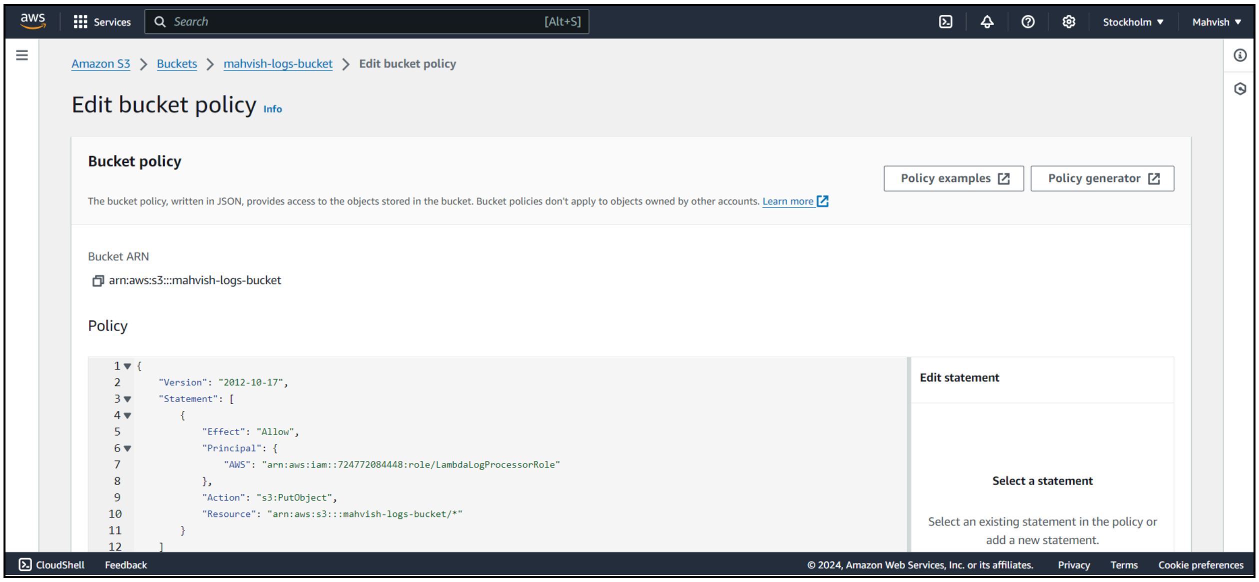
Format: s3://bucket/prefix

Step 6: Go to the permissions tab and add 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:::mahvish-logs-bucket/*"
      }
   ]
}
```

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

"Resource": "arn:aws:s3:::mahvish-logs-bucket/ : this specifies the name of my S3 bucket



Edit bucket policy

Bucket policy

The bucket policy, written in JSON, provides access to the objects stored in the bucket. Bucket policies don't apply to objects owned by other accounts.

Bucket ARN

arn:aws:s3:::mahvish-logs-bucket

Policy

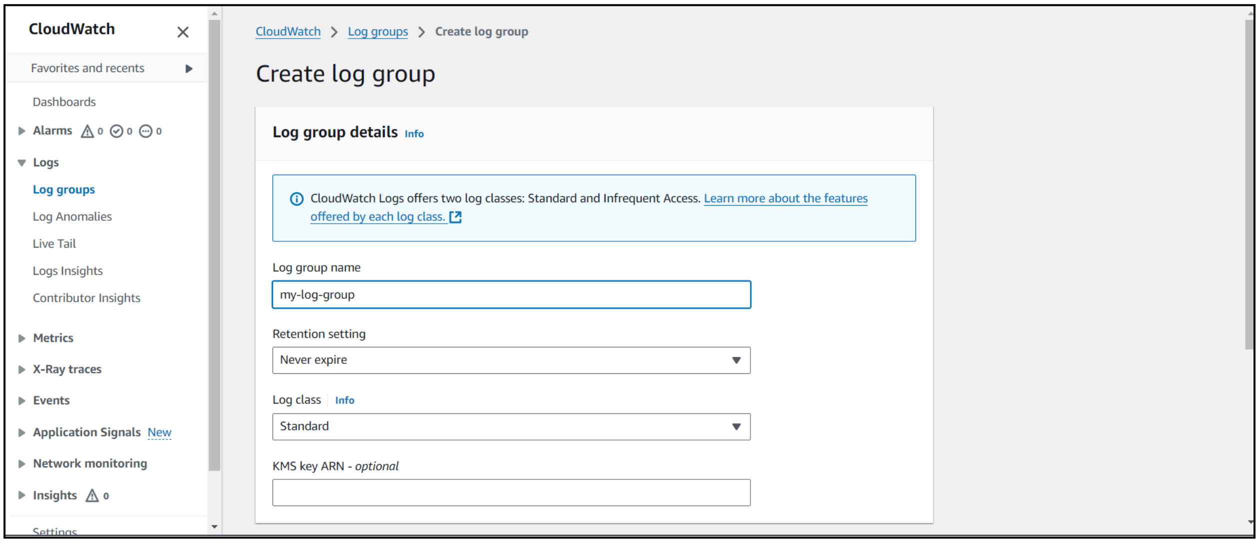
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::724772084448:role/LambdaLogProcessorRole"
            },
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::mahvish-logs-bucket/*"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

Edit statement

Select a statement

Select an existing statement in the policy or add a new statement.

Step 7: Create a cloudswatch log group for saving the logs created by our lambda function



Create log group

Log group details

CloudWatch Logs offers two log classes: Standard and Infrequent Access. Learn more about the features offered by each log class.

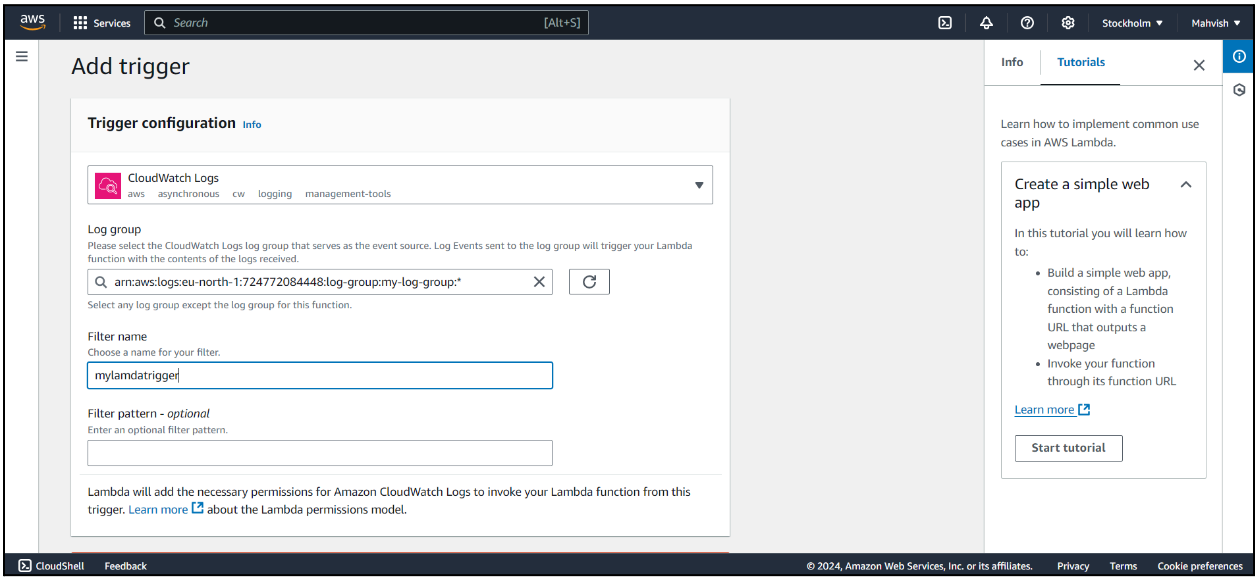
Log group name: my-log-group

Retention setting: Never expire

Log class: Standard

KMS key ARN - *optional*:

Step 8: Now, go back to the lambda function and add the cloudwatch group in triggers



Add trigger

Trigger configuration Info

CloudWatch Logs
aws asynchronous cw logging management-tools

Log group
Please select the CloudWatch Logs log group that serves as the event source. Log Events sent to the log group will trigger your Lambda function with the contents of the logs received.

arn:aws:logs:eu-north-1:724772084448:log-group:my-log-group:*

Select any log group except the log group for this function.

Filter name
Choose a name for your filter.

mylamdatrigger

Filter pattern - *optional*
Enter an optional filter pattern.

Lambda will add the necessary permissions for Amazon CloudWatch Logs to invoke your Lambda function from this trigger. Learn more about the Lambda permissions model.

Learn how to implement common use cases in AWS Lambda.

Create a simple web app

In this tutorial you will learn how to:

- Build a simple web app, consisting of a Lambda function with a function URL that outputs a webpage
- Invoke your function through its function URL

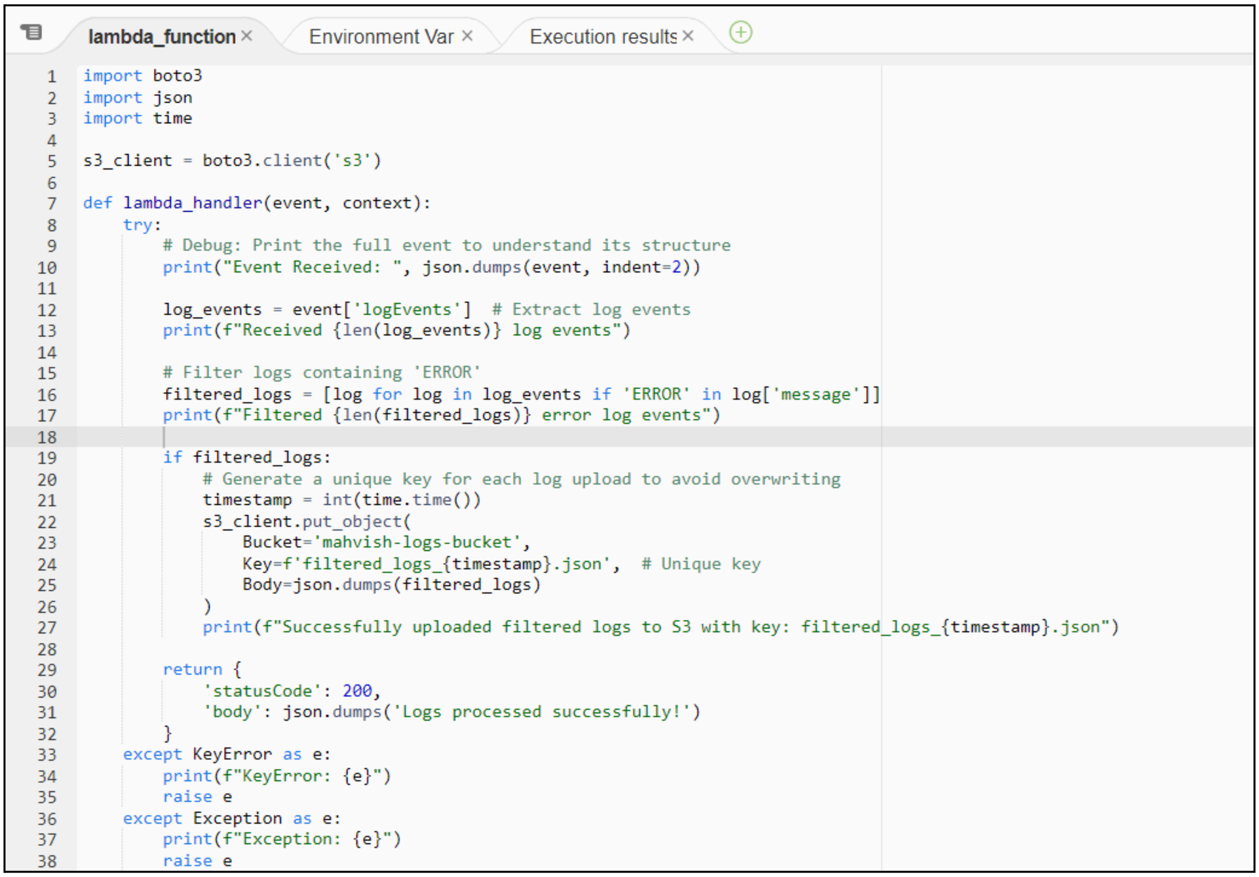
Select the log group created in the previous step

Step 9: Add the code for lambda function

<u>Function code:</u>

```
import boto3
import json
import time

s3_client = boto3.client('s3')

def lambda_handler(event, context):
    try:
        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='mahvish-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
```

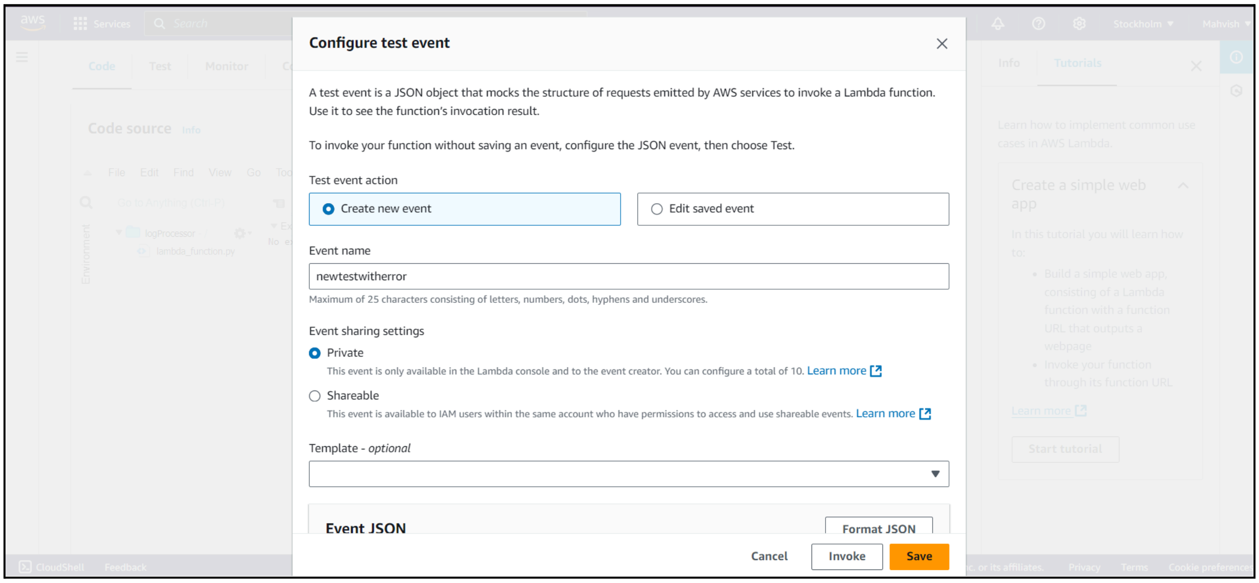
Note: In the above code, add the name of the S3 bucket created earlier.



```
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='mahvish-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
```

After adding the code, click on deploy to save it.

Step 10: Create a new test event to test the setup.



Configure test event

A test event is a JSON object that mocks the structure of requests emitted by AWS services to invoke a Lambda function. Use it to see the function's invocation result.

To invoke your function without saving an event, configure the JSON event, then choose Test.

Test event action

● Create new event ○ Edit saved event

Event name

newtestwitherror

Maximum of 25 characters consisting of letters, numbers, dots, hyphens and underscores.

Event sharing settings

● Private

This event is only available in the Lambda console and to the event creator. You can configure a total of 10. Learn more

○ Shareable

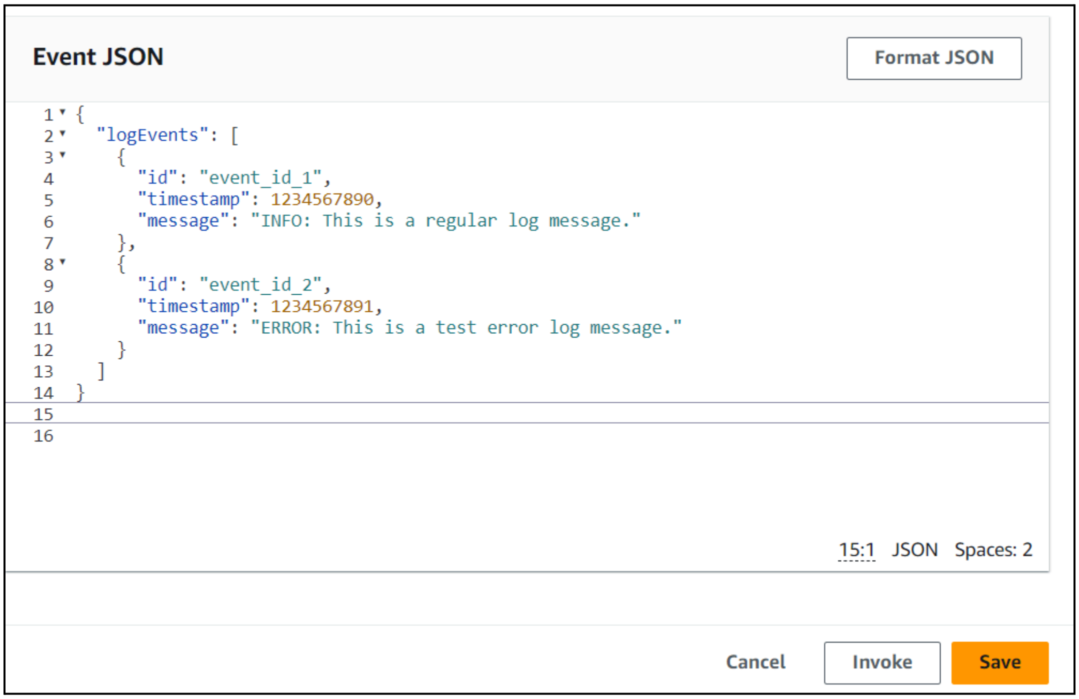
This event is available to IAM users within the same account who have permissions to access and use shareable events. Learn more

Template - *optional*

Event JSON

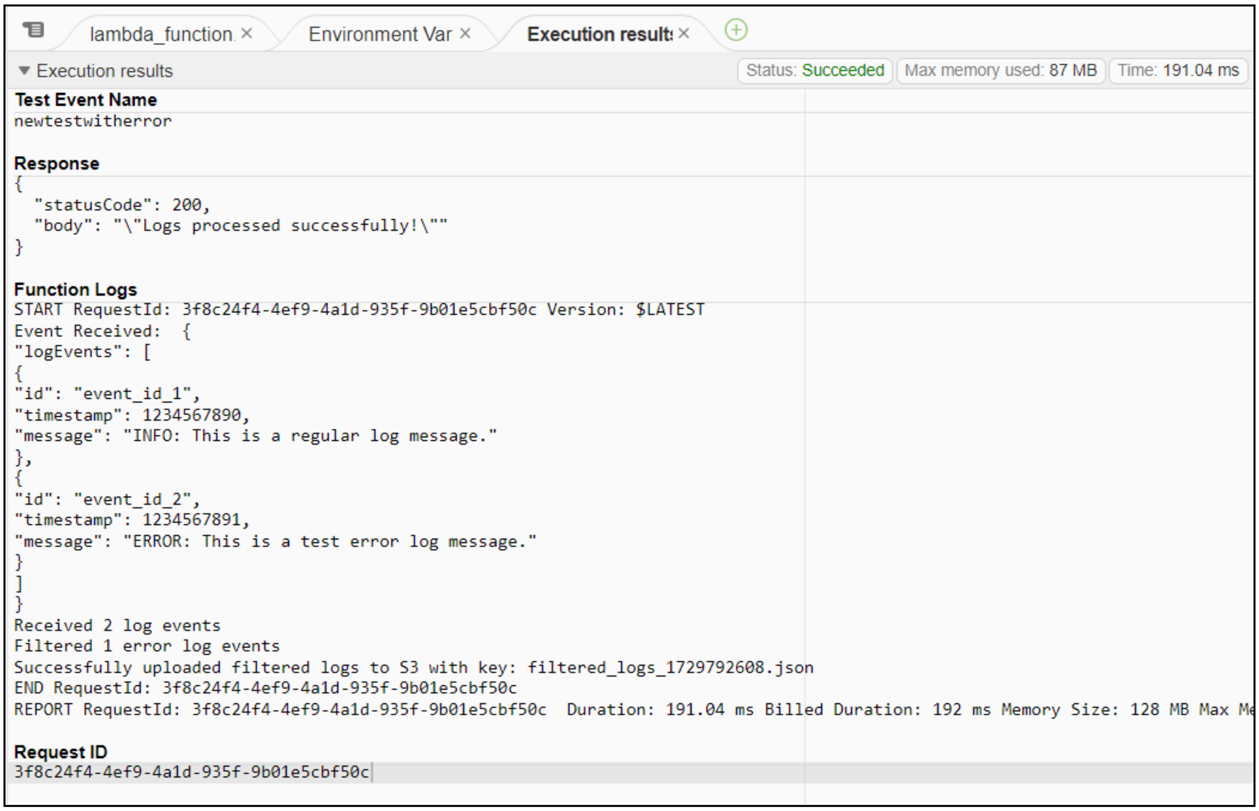
Add this to the json part,

```
{
  "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."
    }
  ]
}
```



```
{
  "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."
    }
  ]
}
```

Click on Invoke to run the test case. A success 200 message appears in the execution results.



Test Event Name

newtestwitherror

Response

```
{
  "statusCode": 200,
  "body": "\"Logs processed successfully!\""
}
```

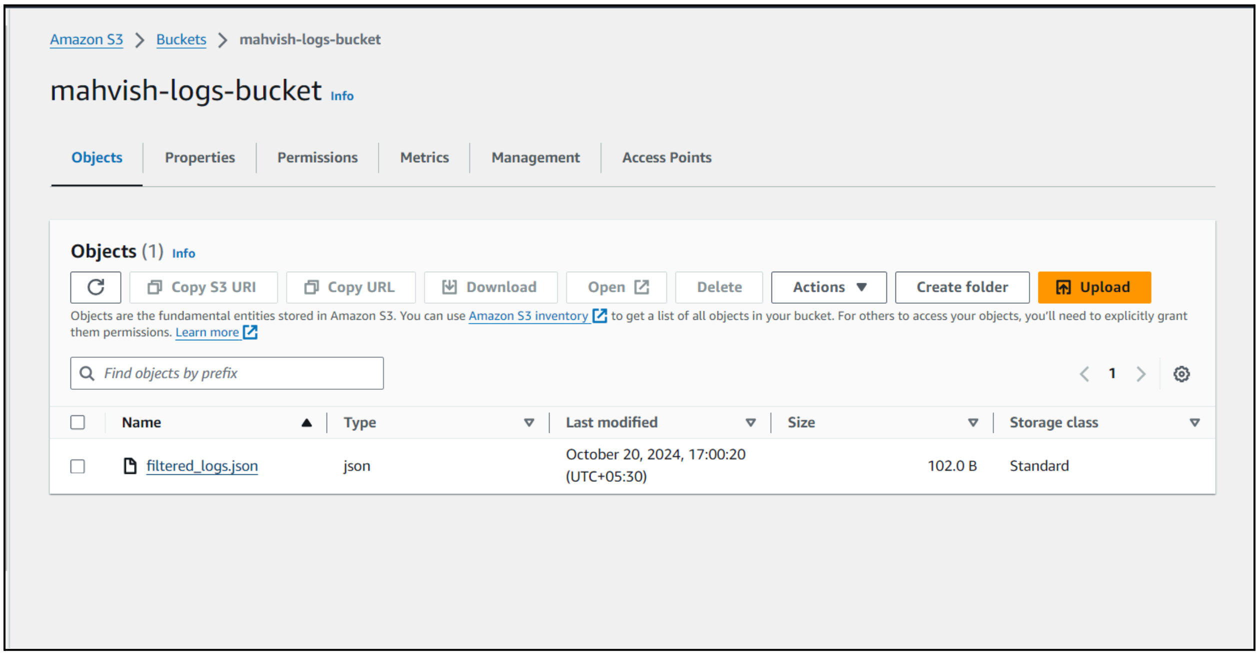
Function Logs

```
START RequestId: 3f8c24f4-4ef9-4a1d-935f-9b01e5cbf50c Version: $LATEST
Event Received:  {
"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."
}
]
}
Received 2 log events
Filtered 1 error log events
Successfully uploaded filtered logs to S3 with key: filtered_logs_1729792608.json
END RequestId: 3f8c24f4-4ef9-4a1d-935f-9b01e5cbf50c
REPORT RequestId: 3f8c24f4-4ef9-4a1d-935f-9b01e5cbf50c  Duration: 191.04 ms Billed Duration: 192 ms Memory Size: 128 MB Max Me
```

Request ID

3f8c24f4-4ef9-4a1d-935f-9b01e5cbf50c

Step 11: To verify, go back to your S3 bucket. A new item called filtered_logs.json is added in the S3 bucket.



Amazon S3 > Buckets > mahvish-logs-bucket

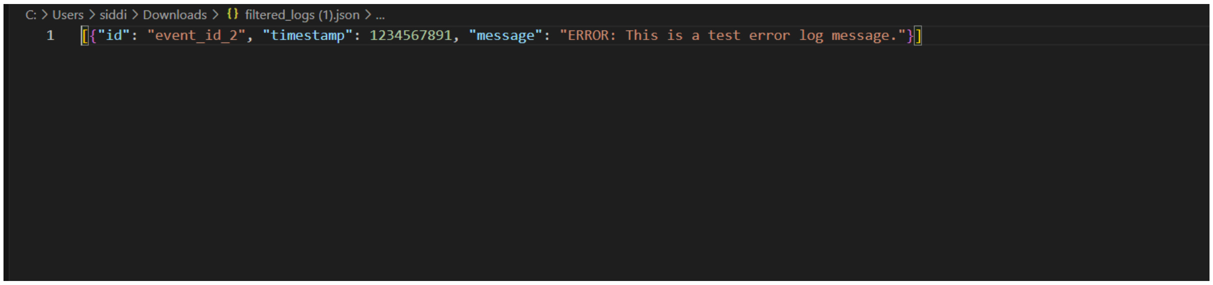
mahvish-logs-bucket Info

Objects (1) Info

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
□	filtered_logs.json	json	October 20, 2024, 17:00:20 (UTC+05:30)	102.0 B	Standard

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



C: > Users > siddi > Downloads > {} filtered_logs (1).json > ...

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

Guidelines:

1. Use your personal AWS account as the AWS academy account does not offer enough privileges to the default role.
2. Principle of Least Privilege: Assign the minimum permissions necessary for IAM roles and policies. This minimizes security risks.
3. Logging and Monitoring: Enable detailed logging for your Lambda functions and monitor them using CloudWatch. This helps in troubleshooting.

Conclusion:

In this case study, the integration of AWS Lambda, CloudWatch Logs, and S3 for real-time log processing was demonstrated. An IAM user with the necessary permissions was created, ensuring secure access to the required AWS services. A Lambda function was set up to trigger on new log entries in a CloudWatch Log Group, filtering specific log events based on a keyword. The filtered logs were then stored in an S3 bucket for further analysis and storage. This system enhanced log monitoring and alerting capabilities while automating the log management process. By following best practices, such as adhering to the principle of least privilege and implementing error handling, a secure, efficient, and scalable solution was achieved.