

**Vivekanand Education Society's Institute of Technology, Chembur, Mumbai,
Department of Technology,
Year: 2024-2025(ODD SEM)**

Advance DevOps Practical Examination

Name: Roma Uday Shirodkar
Roll No: 58

Div: D15B
Date of exam: 24/10/2024

Case Study 18:

Real-Time Log Processing

- **Concepts Used:** AWS Lambda, CloudWatch, S3.
- **Problem Statement:** "Set up a Lambda function that triggers whenever a new log entry is added to a CloudWatch Log Group. The Lambda function should filter specific log events and store them in an S3 bucket."
- **Tasks:**
 - Create a CloudWatch Log Group and set up a Lambda function that triggers on new log entries.
 - Write a Python Lambda function to filter logs based on a keyword (e.g., 'ERROR').
 - Store the filtered logs in an S3 bucket.
 - Test by generating logs and checking the S3 bucket for the filtered entries.

Aim: To set up an AWS Lambda function that triggers on new log entries in a CloudWatch Log Group, filters logs for specific keywords (e.g., 'ERROR'), and stores the filtered logs in an S3 bucket.

Theory:

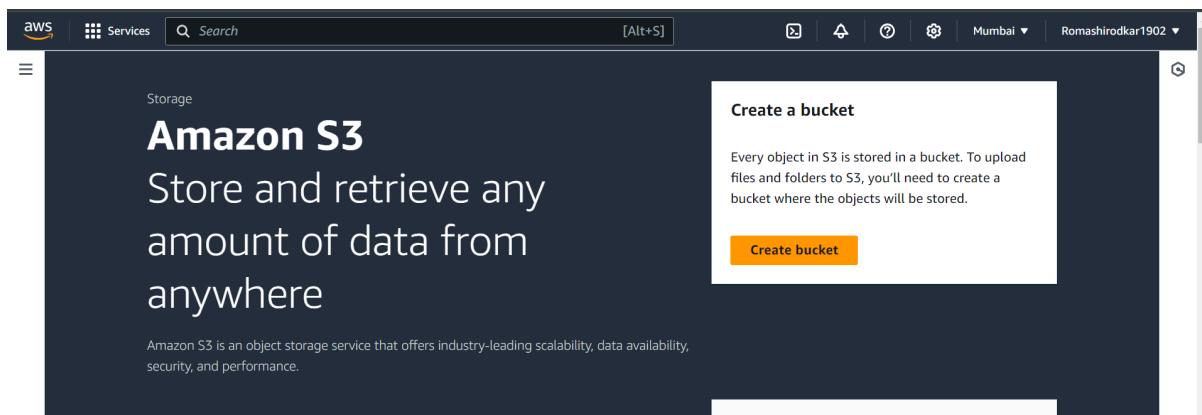
- **Amazon S3** (Simple Storage Service) is a scalable object storage service that allows users to store and retrieve data over the internet. S3 is designed for durability, availability, and performance, making it ideal for a variety of use cases, such as backup and restore, data archiving, big data analytics, and web hosting. *In this experiment, S3 will be used to store filtered log entries generated by the Lambda function.*
- **AWS Lambda** is a serverless compute service that enables users to run code without provisioning or managing servers. It automatically scales to handle incoming requests and only charges for the compute time used. Users can trigger Lambda functions in response to various AWS events, such as changes in data or system state. *In this experiment, a Lambda function will be created to automatically execute when new log entries are added to a CloudWatch Log Group. The function will filter the logs for specific keywords and store the relevant entries in the S3 bucket.*

- **Amazon CloudWatch** is a monitoring and observability service that provides real-time insights into AWS resources and applications. It collects and tracks metrics, logs, and events, allowing users to monitor system performance, troubleshoot issues, and automate responses to changes in resource states. *CloudWatch will be used to create a Log Group that collects log entries from specified sources. The Lambda function will be triggered whenever new log entries are added to this Log Group.*

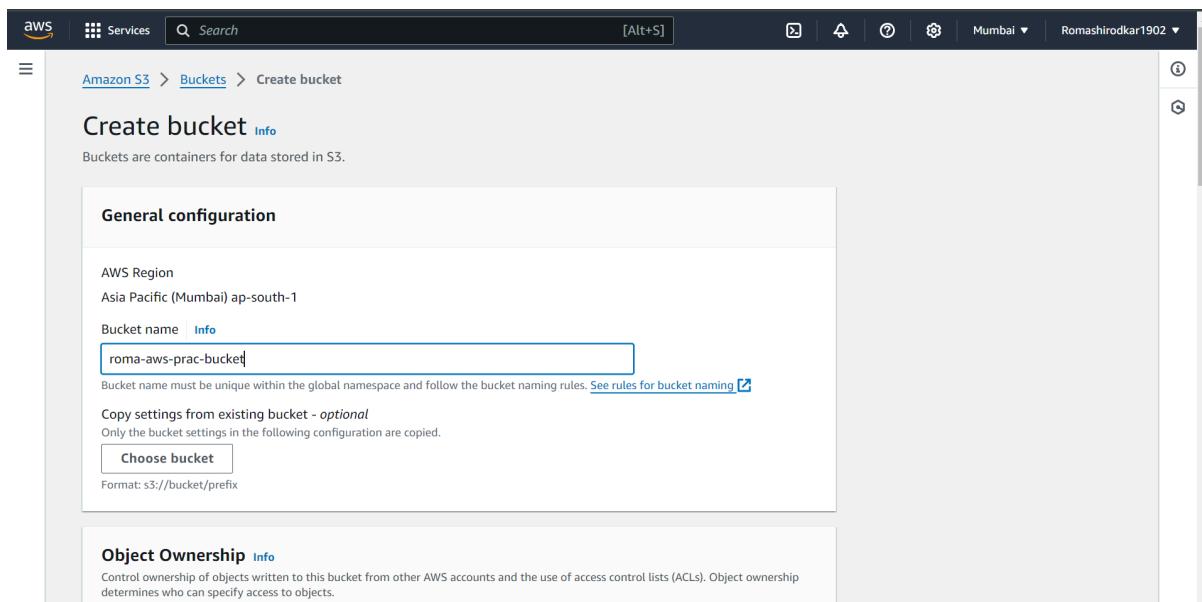
Step-by-Step Implementation:

Step 1: Create an S3 bucket.

- Navigate to the **S3** service in the AWS Management Console, you will see the following page and then click on “Create bucket”.



- Name the bucket. Here I have named it as **roma-aws-prac-bucket**. Keep other options to default and click “create bucket”.



- As we can see here bucket “roma-aws-prac-bucket” has been created.

The screenshot shows the AWS S3 service page. At the top, a green banner indicates that a bucket named "roma-aws-prac-bucket" has been successfully created. Below the banner, there's an "Account snapshot" section with a link to "View details". Under "General purpose buckets", there is one entry:

Name	AWS Region	IAM Access Analyzer	Creation date
roma-aws-prac-bucket	Asia Pacific (Mumbai) ap-south-1	View analyzer for ap-south-1	October 19, 2024, 11:46:06 (UTC+05:30)

Step 2: Create a CloudWatch Log Group

- In the AWS Console, go to CloudWatch.
- On the left panel, click on **Logs** → **Log groups**.
- Click **Create log group**.

The screenshot shows the AWS CloudWatch service page, specifically the "Logs" section. On the left, there's a navigation menu with "Log groups" selected. The main area displays a table for "Log groups (0)". A message at the top states, "By default, we only load up to 10000 log groups." Below the table, it says "No log groups" and "You have not created any log groups." There is a "Create log group" button at the bottom of the table area.

- Provide a name for the log group. Here I have named my log group as **romaAppLogs**. Keep everything else as default and click “Create”.

The screenshot shows the 'Create log group' page in the AWS CloudWatch service. The left sidebar shows 'Logs' selected under 'Logs'. The main form has the following fields:

- Log group details**: Shows a note about log classes and a 'Log group name' input field containing 'romaAppLogs'.
- Retention setting**: Set to 'Never expire'.
- Log class**: Set to 'Standard'.
- KMS key ARN - optional**: An empty input field.

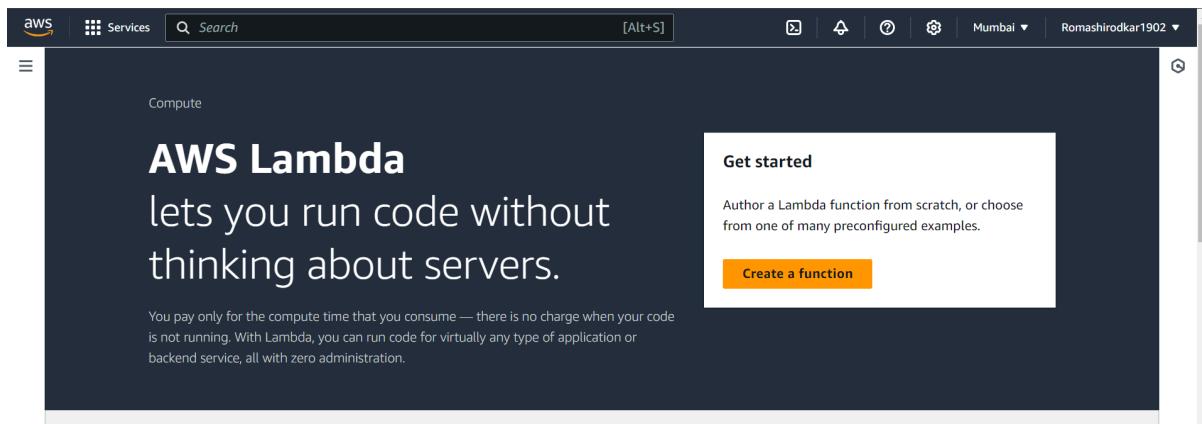
- As we can see below, a log group named **romaAppLogs** has been created.
- All the logs which I will generate in the further steps will be stored in this log group.

The screenshot shows the 'Log groups' page in the AWS CloudWatch service. The left sidebar shows 'Logs' selected under 'Logs'. The main table displays one log group:

Log group	Log class	Anomaly d...	D...	Se...
romaAppLogs	Standard	Configure	-	-

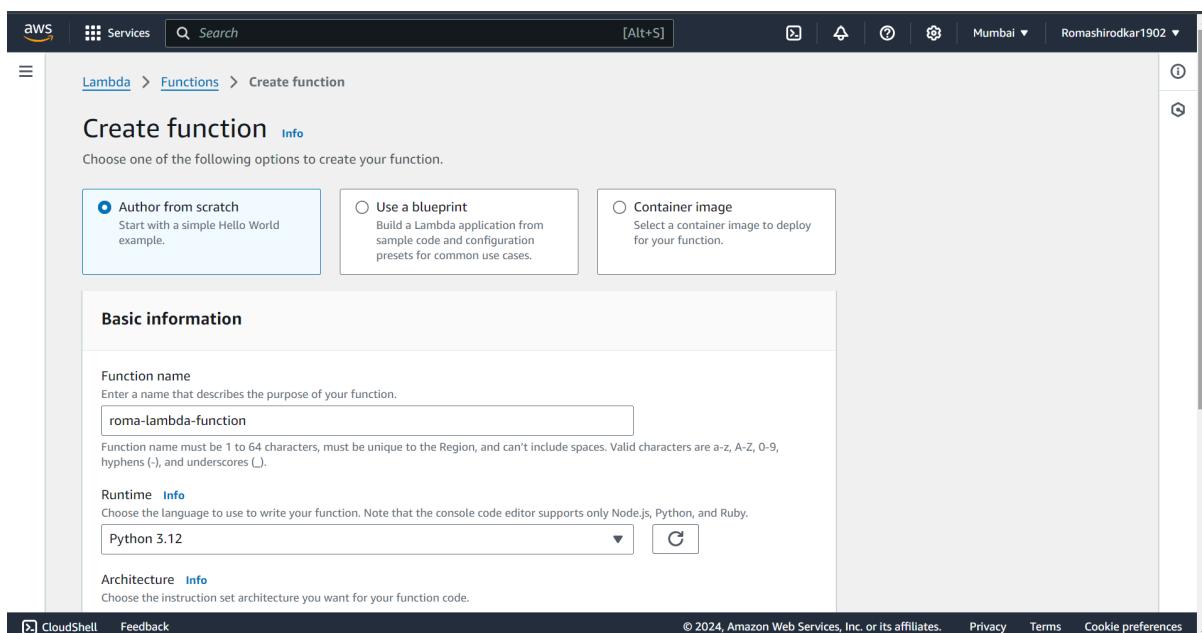
Step 3: Create a Lambda Function

- This is the lambda function which will be triggered every time new logs are generated in “romaAppLogs” log group.
- Go to **AWS Lambda**.
- Click on “Create a function”.

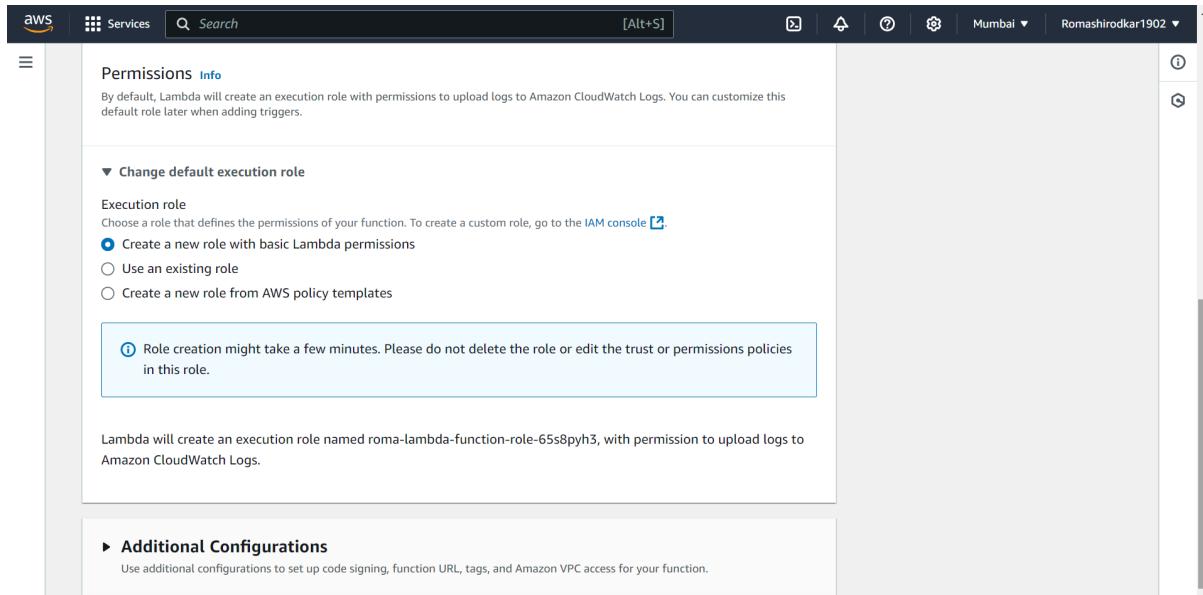


You will see the following page.

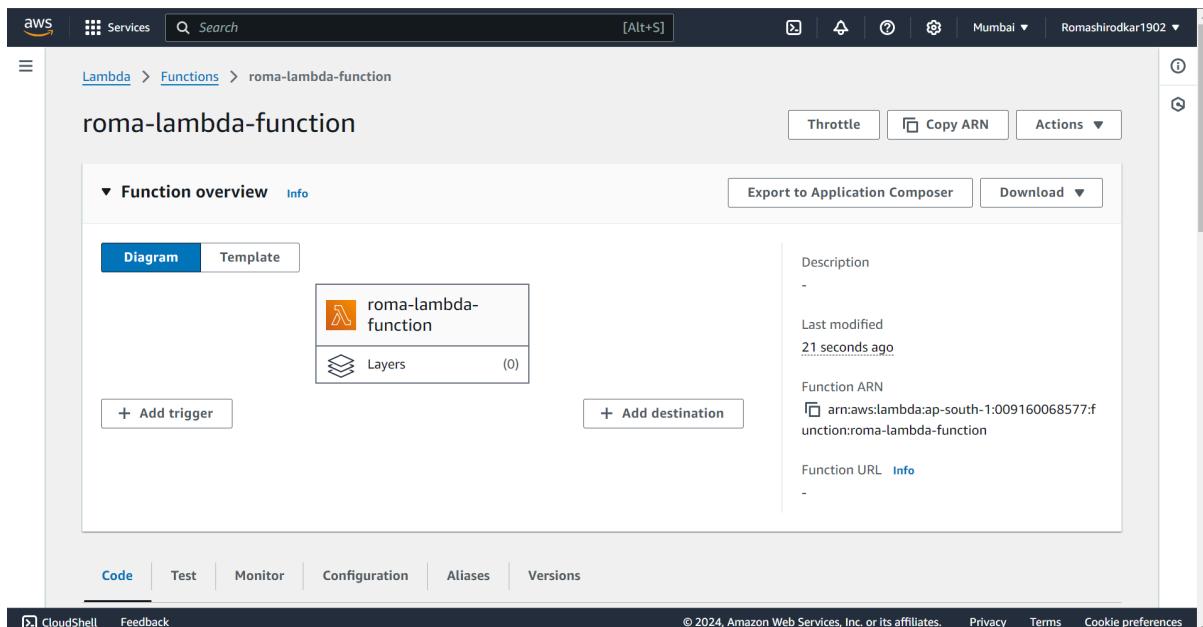
- Select “Author from scratch”.
- Provide a name to the function. Here I have named my function as **roma-lambda-function**.
- Select “Python 3.12” as Runtime.



- Scroll down. You will see the permissions tab open it and make sure “Create a new role with basic Lambda permissions” is selected. This will automatically create a role with permissions to upload logs to Amazon CloudWatch Logs.



This is the Lambda function which I created.



Step 4: Write Python Code for Lambda Function

- Add this code in the code tab of the lambda function.
- This AWS Lambda function processes log data from a CloudWatch Log Group. When invoked, it decodes and decompresses the log data, then filters the log events for entries containing the keyword "ERROR."
- If any such entries are found, they are converted to a JSON string and uploaded to a specified S3 bucket. The function logs its operations, including received events and processing outcomes, for debugging purposes.
- If no error logs are found, it returns a message indicating that there were no errors.

CODE:

```
import boto3
import gzip
import base64
import json
import os

s3 = boto3.client('s3')
bucket_name = 'roma-aws-prac-bucket' # Use your actual S3 bucket name

def lambda_handler(event, context):
    print(f'Received event: {event}')

    if 'awslogs' in event:
        try:
            # Decode and unzip log data
            decoded_data = base64.b64decode(event['awslogs']['data'])
            unzipped_data = gzip.decompress(decoded_data)

            # Parse the log event
            log_data = json.loads(unzipped_data)

            # Filter log events for keyword 'ERROR'
            filtered_logs = [event for event in log_data['logEvents'] if 'ERROR' in event['message']]

            # If no errors, skip processing
            if not filtered_logs:
                print("No ERROR logs found.")
                return {
                    'statusCode': 200,
                    'body': json.dumps('No errors found')
                }

            # Convert filtered logs to string
            filtered_logs_str = json.dumps(filtered_logs)

            # Define the S3 bucket and object key
            object_key = f"filtered_logs_{context.aws_request_id}.json"
            s3.put_object(Bucket=bucket_name, Key=object_key, Body=filtered_logs_str)

        except Exception as e:
            print(f'Error processing log: {e}')

    return {
        'statusCode': 200,
        'body': json.dumps('Lambda function executed successfully')
    }
```

```

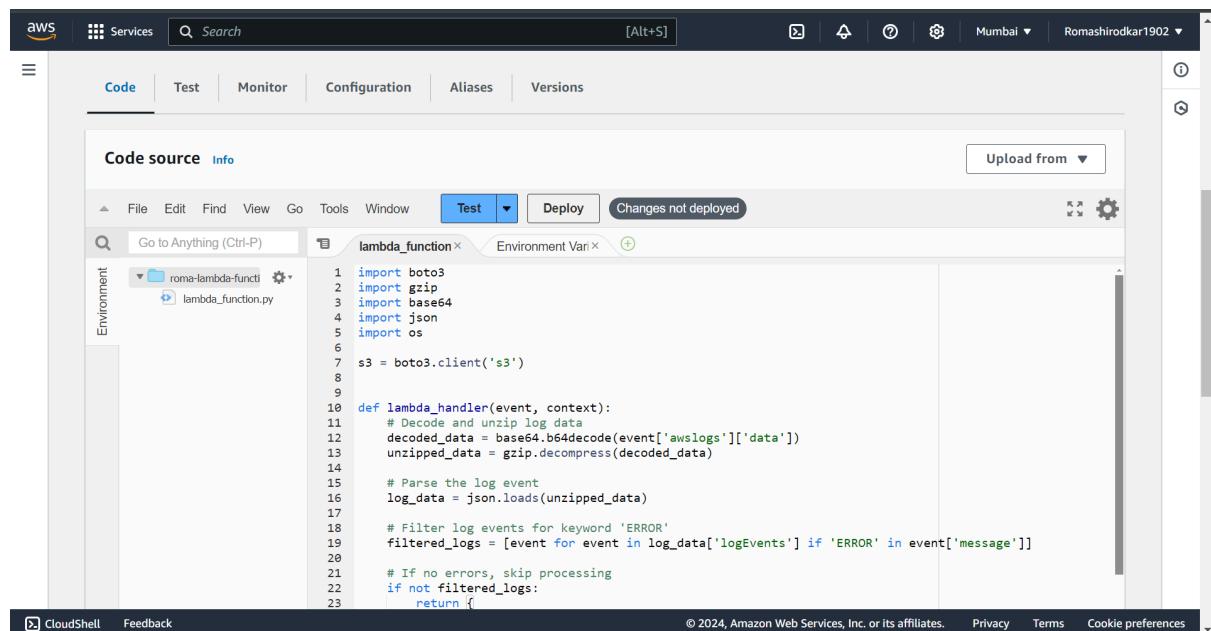
# Upload filtered logs to S3
s3.put_object(Bucket=bucket_name, Key=object_key, Body=filtered_logs_str)
print(f'Filtered logs uploaded to S3: {object_key}')

return {
    'statusCode': 200,
    'body': json.dumps('Filtered logs uploaded to S3')
}

except Exception as e:
    print(f'Error processing log data: {e}')
    return {
        'statusCode': 500,
        'body': json.dumps('Error processing log data')
    }
else:
    print("No awslogs found in event.")
    return {
        'statusCode': 400,
        'body': json.dumps('Invalid event structure')
    }

```

I have added the above code in the lambda_function.py file



The screenshot shows the AWS Lambda console interface. The top navigation bar includes 'Services', 'Search', 'Mumbai', and 'Romashirodkar1902'. Below the navigation is a toolbar with 'Code', 'Test', 'Monitor', 'Configuration', 'Aliases', and 'Versions'. The 'Code' tab is selected. The main area is titled 'Code source' with tabs for 'Info' and 'Upload from'. A 'Test' dropdown shows 'Changes not deployed'. The code editor displays the Python script 'lambda_function.py' with the following content:

```

1 import boto3
2 import gzip
3 import base64
4 import json
5 import os
6
7 s3 = boto3.client('s3')
8
9
10 def lambda_handler(event, context):
11     # Decode and unzip log data
12     decoded_data = base64.b64decode(event['awslogs']['data'])
13     unzipped_data = gzip.decompress(decoded_data)
14
15     # Parse the log event
16     log_data = json.loads(unzipped_data)
17
18     # Filter log events for keyword 'ERROR'
19     filtered_logs = [event for event in log_data['logEvents'] if 'ERROR' in event['message']]
20
21     # If no errors, skip processing
22     if not filtered_logs:
23         return {

```

- The code provided contains one environment variable, S3_BUCKET_NAME, which is intended to hold the name of the S3 bucket where the filtered logs will be stored.

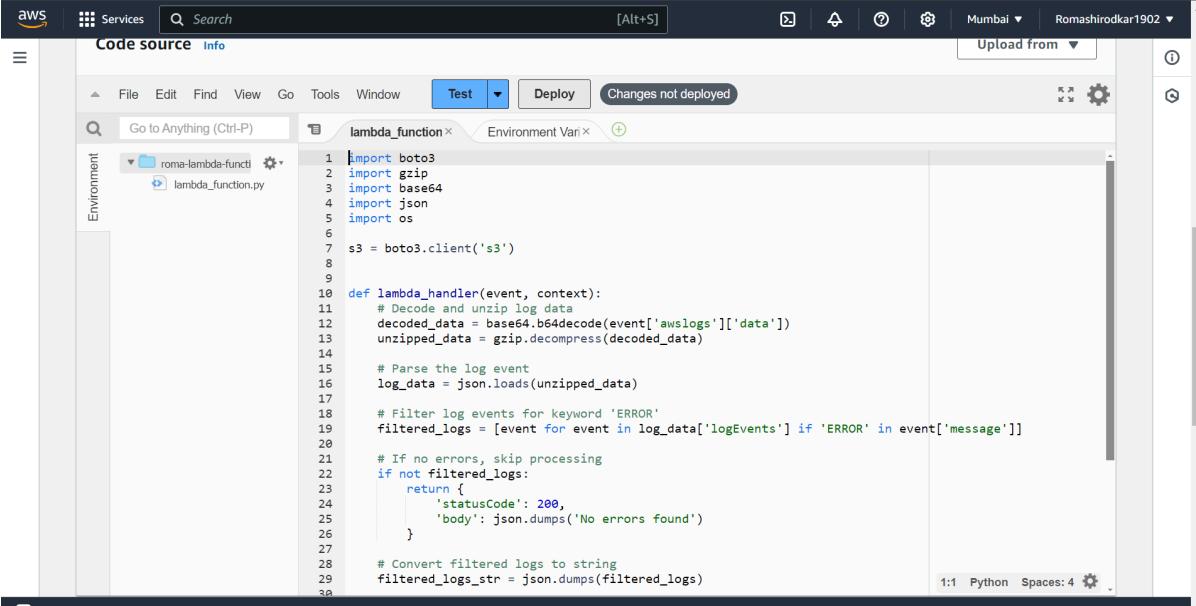
- Go to Configuration → Environment Variables
- Click on Edit

The screenshot shows the AWS Lambda configuration interface. The left sidebar lists various configuration options: General configuration, Triggers, Permissions, Destinations, Function URL, Environment variables (which is currently selected), Tags, VPC, RDS databases, Monitoring and operations tools, and Concurrency and recursion detection. The main content area is titled 'Environment variables (0)' and contains a search bar with placeholder text 'Find environment variables'. Below the search bar is a table with two columns: 'Key' and 'Value'. A message in the center of the table states 'No environment variables associated with this function.' At the bottom right of the table is a small 'Edit' button.

- In key, type “S3_BUCKET_NAME”
- In value, enter the s3 bucket name. I have added my s3 bucket’s name i.e. **Roma_aws_prac_bucket** and then save it.

The screenshot shows the 'Edit environment variables' dialog box. It has a header 'Edit environment variables' and a sub-header 'Environment variables'. A note below the sub-header explains that environment variables are key-value pairs used to store configuration settings. The main area contains a table with two columns: 'Key' and 'Value'. A single row is present with 'S3_BUCKET_NAME' in the 'Key' column and 'roma_aws_prac_bucket' in the 'Value' column. Below the table are two buttons: 'Add environment variable' and 'Remove'. At the bottom of the dialog are 'Cancel' and 'Save' buttons, with 'Save' being the active button.

- Save your function (ctrl+S) and click Deploy.



```

1 import boto3
2 import gzip
3 import base64
4 import json
5 import os
6
7 s3 = boto3.client('s3')
8
9
10 def lambda_handler(event, context):
11     # Decode and unzip log data
12     decoded_data = base64.b64decode(event['awslogs']['data'])
13     unzipped_data = gzip.decompress(decoded_data)
14
15     # Parse the log event
16     log_data = json.loads(unzipped_data)
17
18     # Filter log events for keyword 'ERROR'
19     filtered_logs = [event for event in log_data['logEvents'] if 'ERROR' in event['message']]
20
21     # If no errors, skip processing
22     if not filtered_logs:
23         return {
24             'statusCode': 200,
25             'body': json.dumps('No errors found')
26         }
27
28     # Convert filtered logs to string
29     filtered_logs_str = json.dumps(filtered_logs)
30

```

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

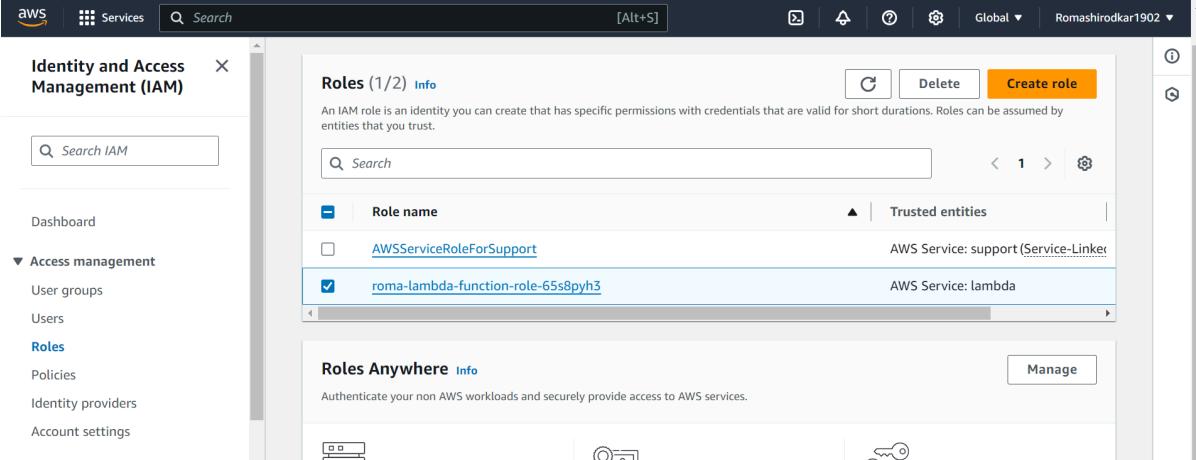
Step 5: Set Permissions for Lambda

Go to IAM in the AWS Console.

Find the role created for your Lambda function.

Attach the following managed policies:

- AWSLambdaBasicExecutionRole
- CloudWatchLogsFullAccess
- AmazonS3FullAccess (or a custom S3 policy with write access to the specific bucket).



Role Name	Trusted entities
AWSServiceRoleForSupport	AWS Service: support (Service-Linked)
roma-lambda-function-role-65s8pyh3	AWS Service: lambda

Screenshot of the AWS IAM Permissions page.

Left sidebar:

- Identity and Access Management (IAM)
- Dashboard
- Access management
 - User groups
 - Users
 - Roles**
 - Policies
 - Identity providers
 - Account settings
- Access reports
- Access Analyzer

Main content area:

Last activity: - Maximum session duration: 1 hour

Permissions tab selected. Other tabs: Trust relationships, Tags, Last Accessed, Revoke sessions.

Permissions policies (1) [Info](#)

You can attach up to 10 managed policies.

Filter by Type: All types

Policy name	Type	Attached entities
AWSLambdaBasicExecution...	Customer managed	1

Actions: [C](#) [Simulate](#) [Remove](#) [Add permissions](#) [Attach policies](#) [Create inline policy](#)

Permissions boundary (not set)

Screenshot of the Attach policy to role page.

Path: IAM > Roles > [roma-lambda-function-role-65s8pyh3](#) > Add permissions

Attach policy to **roma-lambda-function-role-65s8pyh3**

Current permissions policies (1)

Other permissions policies (2/958)

Filter by Type: All types

Policy name	Type	Description
CloudWatchLogsFullAccess	AWS managed	Provides full access to CloudWatch Logs

Actions: [C](#) [Cancel](#) [Add permissions](#)

Screenshot of the Attach policy to role page showing a search result.

Path: IAM > Roles > [roma-lambda-function-role-65s8pyh3](#) > Add permissions

Attach policy to **roma-lambda-function-role-65s8pyh3**

Current permissions policies (1)

Other permissions policies (2/958)

Filter by Type: All types

Search results for "S3": 9 matches

Policy name	Type	Description
AmazonDMSRedshiftS3Role	AWS managed	Provides access to manage S3 settings ...
AmazonS3FullAccess	AWS managed	Provides full access to all buckets via t...
AmazonS3ObjectLambdaExecutionRoleP...	AWS managed	Provides AWS Lambda functions permi...
AmazonS3OutpostsFullAccess	AWS managed	Provides full access to Amazon S3 on ...
AmazonS3OutpostsReadOnlyAccess	AWS managed	Provides read only access to Amazon S...

The screenshot shows the AWS Identity and Access Management (IAM) service. In the left sidebar, under 'Access management', 'Roles' is selected. The main content area displays 'Permissions policies (3)'. The table lists three policies:

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

Step 6: Create a CloudWatch Log Group Trigger

Navigate to lambda function created and click on “Add trigger”

The screenshot shows the AWS Lambda service. The function name is 'roma-lambda-function'. Under the 'Function overview' section, there is a 'Diagram' tab which is currently selected, showing a single function icon labeled 'roma-lambda-function'. Below the diagram is a '+ Add trigger' button. To the right, there is a 'Description' field, 'Last modified' (35 minutes ago), 'Function ARN' (arn:aws:lambda:ap-south-1:009160068577:function:roma-lambda-function), and 'Function URL'.

Select **CloudWatch Logs** as the source.

The screenshot shows the 'Add trigger' dialog in the AWS Lambda service. The 'Trigger configuration' section has a search bar where 'Cloud' is typed. The results show 'CloudWatch Logs' under 'Batch/bulk data processing'. The 'CloudWatch Logs' option is highlighted with a blue selection bar. At the bottom right of the dialog are 'Cancel' and 'Add' buttons.

Provide a filter name and also choose the log group created earlier.

Trigger configuration [Info](#)

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.

Filter name

Choose a name for your filter.

roma-aws-prac-filter

Filter pattern - optional

Step 7: Generate Logs for testing

Open AWS CLI and enter the following command

```
aws logs create-log-stream --log-group-name romaAppLogs --log-stream-name romaAppStream.
```

This will create a Log stream named romaAppStream for romaAppLogs log group

```
[cloudshell-user@ip-10-130-25-231 ~]$ aws logs create-log-stream --log-group-name romaAppLogs --log-stream-name romaAppStream
[cloudshell-user@ip-10-130-25-231 ~]$
```

CloudWatch

Favorites and recent

CloudShell

Log groups

Log Anomalies

Live Tail

Logs Insights

Contributor Insights

Metrics

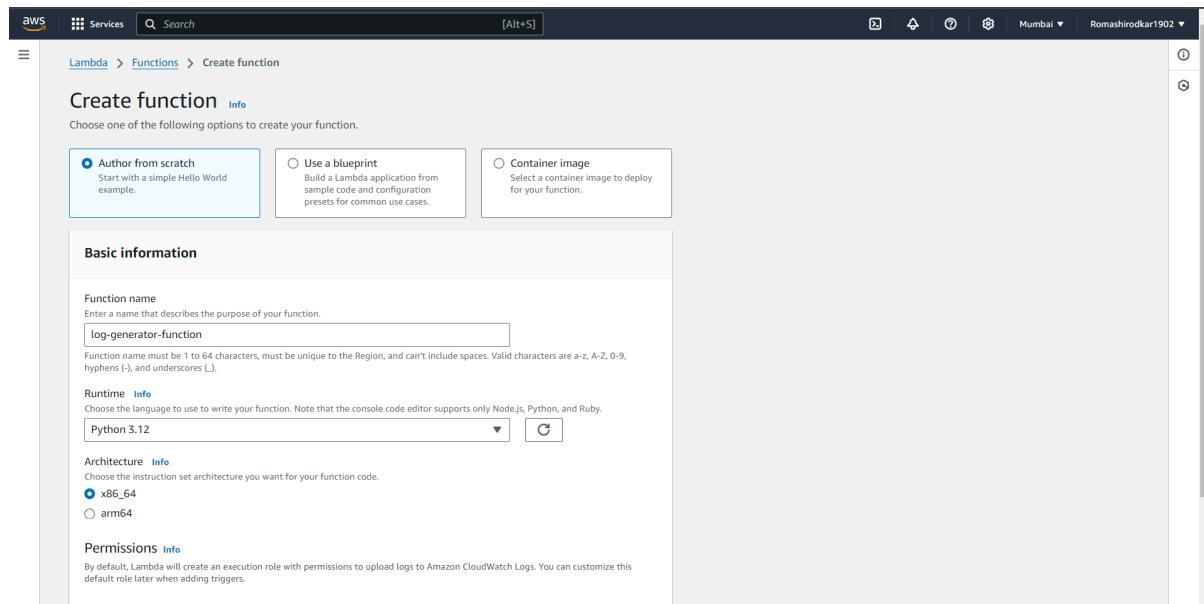
X-Ray traces

Events

Log streams (1)

Log stream	Last event time
romaAppStream	-

Now we will create another lambda function to generate logs



Add this code

```
import boto3
import random
import json
import time

# Initialize the CloudWatch Logs client
logs_client = boto3.client('logs')

# Define your log group and log stream names
LOG_GROUP_NAME = 'romaAppLogs' # Your existing log group name
LOG_STREAM_NAME = 'romaAppStream' # Your existing log stream name

def lambda_handler(event, context):
    # Get the sequence token for the log stream
    try:
        response = logs_client.describe_log_streams(
            logGroupName=LOG_GROUP_NAME,
            logStreamNamePrefix=LOG_STREAM_NAME
        )
        log_stream = response['logStreams'][0]
        sequence_token = log_stream.get('uploadSequenceToken', None)
    except Exception as e:
        print(f"Error retrieving sequence token: {e}")
    return {
        'statusCode': 500,
```

```

        'body': json.dumps('Error retrieving sequence token')
    }

# Start continuous log generation (run for 14 minutes)
end_time = time.time() + (14 * 60) # 14 minutes loop (as Lambda timeout is max 15 minutes)

while time.time() < end_time:
    # Generate a random log message
    log_message = {
        'timestamp': int(time.time() * 1000),
        'message': random.choice(['ERROR: Something went wrong!', 'SUCCESS: Operation completed successfully.'])
    }

    # Prepare the parameters for putting log events
    put_log_events_args = {
        'logGroupName': LOG_GROUP_NAME,
        'logStreamName': LOG_STREAM_NAME,
        'logEvents': [log_message]
    }

    # Include the sequence token if it exists
    if sequence_token:
        put_log_events_args['sequenceToken'] = sequence_token

    # Send the log event to the log stream
    try:
        response = logs_client.put_log_events(**put_log_events_args)
        sequence_token = response['nextSequenceToken'] # Update the sequence token for the next call
        print("Log event sent successfully.")
    except Exception as e:
        print(f"Error sending log event: {e}")
        return {
            'statusCode': 500,
            'body': json.dumps('Error sending log event')
        }

    # Pause for a while before generating the next log (e.g., every 5 seconds)
    time.sleep(5)

return {
    'statusCode': 200,
    'body': json.dumps('Continuous log generation complete!')
}

```

```

1 import boto3
2 import random
3 import json
4 import time
5
6 # Initialize the CloudWatch Logs client
7 logs_client = boto3.client('logs')
8
9 # Define your log group and log stream names
10 LOG_GROUP_NAME = 'romaAppLogs' # Your existing log group name
11 LOG_STREAM_NAME = 'romaAppStream' # Your existing log stream name
12
13 def lambda_handler(event, context):
14     # Generate a random log message
15     log_message = {
16         'timestamp': int(time.time() * 1000),
17         'message': random.choice(['ERROR: Something went wrong!', 'SUCCESS: Operation completed successfully.'])
18     }
19
20     # Put the log message to the existing log stream
21     try:
22         logs_client.put_log_events(
23             logGroupName=LOG_GROUP_NAME,
24             logStreamName=LOG_STREAM_NAME,
25             logEvents=[log_message]
26         )
27         print("Log event sent successfully.")
28     except Exception as e:
29         print(f"Error sending log event: {e}")
30
31     return {
32         'statusCode': 200,
33         'body': json.dumps('Log generation complete!')
34     }

```

This function will need permission to put logs in CloudWatch log groups.
So Navigate to IAM console and attach policies.

Permissions policies (1) Info	
You can attach up to 10 managed policies.	
<input type="button" value="C"/> <input type="button" value="Simulate"/> <input type="button" value="Remove"/> <input type="button" value="Add permissions"/>	
Filter by Type <input type="text" value="Search"/> <input type="button" value="All types"/>	

The screenshot shows the AWS IAM service interface. In the top navigation bar, there are tabs for 'Services' and a search bar. On the right, there are global settings and user information (Romashirodkar1902). Below the navigation, a sidebar lists 'Current permissions policies (1)' and 'Other permissions policies (1/969)'. A search bar at the top of the main content area is set to 'CloudWatchLogs'. A table below shows a list of policies, with one policy selected:

Policy name	Type	Description
AmazonAPIGatewayPushToCloudWatchLogs	AWS managed	Allows API Gateway to push logs to us...
AmazonDMSCloudWatchLogsRole	AWS managed	Provides access to upload DMS replicat...
AWSAppSyncPushToCloudWatchLogs	AWS managed	Allows AppSync to push logs to user's ...
AWSOpsWorksCloudWatchLogs	AWS managed	Enables OpsWorks instances with the ...
CloudWatchLogsCrossAccountSharingCo...	AWS managed	Provides capabilities to manage Obser...
<input checked="" type="checkbox"/> CloudWatchLogsFullAccess	AWS managed	Provides full access to CloudWatch Logs
CloudWatchLogsReadOnlyAccess	AWS managed	Provides read only access to CloudWat...

At the bottom right of the table are 'Cancel' and 'Add permissions' buttons.

To automatically trigger this function we create a scheduler in Amazon EventBridge

The screenshot shows the AWS CloudWatch service interface with a search bar for 'amazon eventBridge'. The left sidebar is titled 'CloudWatch' and includes sections for Favorites and recent items, Logs, Metrics, X-Ray traces, Events, Application Signals, Network monitoring, Insights, and general settings. The main content area displays the 'Amazon EventBridge' service details, including its description as a serverless service for building event-driven applications, its top features (Event buses, Rules, Partner event sources, Schemas, Pipes), and related services like Amazon EventBridge Scheduler, Elastic Container Service, and AWS Private Certificate Authority. There are also sections for Features (Dashboard, Export snapshots to EC2) and a sidebar for configuring Application Insights.

Click on Create Rule

The screenshot shows the AWS EventBridge landing page. On the left, there's a sidebar with navigation links like Dashboard, Developer resources, Buses, Pipes, Scheduler, Integration, and more. The main content area features a video thumbnail for 'Serverless 101: Amazon EventBridge'. To the right, there are sections for 'Get started' (with options for EventBridge Rule, Pipes, Schedule, Schema registry), 'Benefits and features' (listing Build event-driven architectures, Write less custom code, Reduce and secure, and Faster time to production), and 'Pricing' (noting no up-front commitment or minimum fee). A prominent orange 'Create rule' button is located in the 'Get started' section.

Provide details such as name for scheduler

The screenshot shows the 'Specify schedule detail' step of the AWS EventBridge scheduler creation wizard. It's Step 1 of 4. The form includes fields for 'Schedule name and description' (Schedule name: 'log-generator', Description: 'Enter description'), and a 'Schedule group' dropdown set to 'default'. Other steps in the wizard are visible on the left: Step 2 (optional) 'Select target', Step 3 (optional) 'Settings', and Step 4 'Review and create schedule'.

Select rate based scheduler and set (1 minutes) as rate

The screenshot shows the 'Schedule pattern' configuration page for AWS Lambda. The 'Recurring schedule' option is selected. The time zone is set to '(UTC+05:30) Asia/Calcutta'. The 'Rate-based schedule' section is selected, showing a rate of '1 minutes'. A 'Flexible time window' dropdown is present. The bottom navigation bar includes CloudShell, Feedback, Privacy, Terms, and Cookie preferences.

In target select Lambda and select log-generator-function.

The screenshot shows the 'Select target' configuration page for AWS Lambda. The 'Templated targets' option is selected. The 'AWS Lambda Invoke' target is selected. The bottom navigation bar includes CloudShell, Feedback, Privacy, Terms, and Cookie preferences.

The screenshot shows two stacked configuration pages from the AWS console.

The top section is the "Invoke" configuration for AWS Lambda. It includes fields for "Lambda function" (set to "log-generator-function"), "Payload" (containing the JSON object {"1": 1}), and "Universal target definition".

The bottom section is the "EventBridge Scheduler" configuration. It shows options for selecting an Amazon SQS queue as a DLQ (None selected), encryption settings (Info: "Your data is encrypted by default with a key that AWS owns and manages for you. To choose a different key, customise your encryption settings."), and permissions (Execution role: "Create new role for this schedule" selected). A "Next" button is visible at the bottom right.

This scheduler needs permission to execute function.

The screenshot shows the "Identity and Access Management (IAM)" page, specifically the "Roles" section. A role named "Amazon_EventBridge_Scheduler_LAMBDA_6de5ee2cbd" is selected.

The "Summary" tab displays the ARN of the role: "arn:aws:iam::009160068577:role/service-role/Amazon_EventBridge_Scheduler_LAMBDA_6de5ee2cbd".

The "Permissions" tab shows three managed policies attached to the role: "AWSLambdaBasicExecutionRole", "AWSLambdaVPCAccessExecutionRole", and "AWSLambdaPowerInvokeExecutionRole".

Attach the following policies

Policy name	Type	Description
AmazonS3ObjectLambdaExecutionRole...	AWS managed	Provides AWS Lambda functions permi...
AmazonSageMakerPartnerServiceCatalog...	AWS managed	Service role policy used by the AWS La...
AmazonSageMakerServiceCatalogProduct...	AWS managed	Service role policy used by the AWS La...
AWSCodeDeployRoleForLambda	AWS managed	Provides CodeDeploy service access to ...
AWSCodeDeployRoleForLambdaLimited	AWS managed	Provides CodeDeploy service limited a...
AWSDeepLensLambdaFunctionAccessPol...	AWS managed	This policy specifies permissions requi...
AWSLambda_FullAccess	AWS managed	Grants full access to AWS Lambda serv...
AWSLambda_ReadOnlyAccess	AWS managed	Grants read-only access to AWS Lamb...
AWSLambdaBasicExecutionRole	AWS managed	Provides write permissions to CloudW...
AWSLambdaBasicExecutionRole-072128ad...	Customer managed	-
AWSLambdaBasicExecutionRole-14565cf8...	Customer managed	-

After that if we check CloudWatch Logs and S3 bucket we see the following.

Timestamp	Message
2024-10-24T02:44:20,475Z	ERROR: Something went wrong!

AWS Services Search [Alt+S] Mumbai Romashirodkar1902

Amazon S3

Buckets Access Grants Access Points Object Lambda Access Points Multi-Region Access Points Batch Operations IAM Access Analyzer for S3

Block Public Access settings for this account

Storage Lens Dashboards Storage Lens groups AWS Organizations settings

Feature spotlight 7

AWS Marketplace for S3

Objects (1) Info

Upload

Find objects by prefix

Name	Type	Last modified	Size	Storage class
filtered_logs_5692362a-2ca9-4597-8bee-6b7edcb0673c.json	json	October 24, 2024, 08:14:23 (UTC+05:30)	139.0 B	Standard

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

AWS Services Search event Subscriptions Mumbai Romashirodkar1902

CloudWatch

Favorites and recent Dashboards Alarms 0 0 0 Logs Log groups Log Anomalies Live Tail Logs Insights Contributor Insights Metrics X-Ray traces Events Rules Event Buses Application Signals Network monitoring Insights 0

Log events

CloudWatch > Log groups > romaAppLogs > romaAppStream

Actions Start tailing Create metric filter

Filter events - press enter to search Clear 1m 30m 1h 12h Custom UTC timezone Display

Timestamp	Message
No older events at this moment. Retry	
2024-10-24T02:44:20.475Z	ERROR: Something went wrong!
2024-10-24T02:52:24.873Z	SUCCESS: Operation completed successfully.
No newer events at this moment. Auto retry paused. Resume	

AWS Services Search [Alt+S] Mumbai Romashirodkar1902

CloudWatch

Favorites and recent Dashboards Alarms 0 0 0 Logs Log groups Log Anomalies Live Tail Logs Insights Contributor Insights Metrics X-Ray traces Events Rules Event Buses Application Signals Network monitoring Insights 0

Log events

CloudWatch > Log groups > romaAppLogs > romaAppStream

Actions Start tailing Create metric filter

Filter events - press enter to search Clear 1m 30m 1h 12h Custom UTC timezone Display

Timestamp	Message
No older events at this moment. Retry	
2024-10-24T02:44:20.475Z	ERROR: Something went wrong!
2024-10-24T02:52:24.873Z	SUCCESS: Operation completed successfully.
2024-10-24T03:00:04.000Z	ERROR: Failed to connect to database
No newer events at this moment. Auto retry paused. Resume	

Objects (2) Info

Name	Type	Last modified	Size	Storage class
filtered_logs_30d67e03-0b39-4083-a426-f895614f9c6a.json	json	October 24, 2024, 08:30:08 (UTC+05:30)	147.0 B	Standard
filtered_logs_5692362a-2ca9-4597-8bee-6b7edcb0673c.json	json	October 24, 2024, 08:14:23 (UTC+05:30)	139.0 B	Standard

```

filtered_logs_30d67e03-0b39-4083-a426-f895614f9c6a.json
C:\Users\udays\Downloads> filtered_logs_30d67e03-0b39-4083-a426-f895614f9c6a.json
1 [{"id": "38574464328015102632305841952468229161525849040916905984", "timestamp": 1729738804000, "message": "ERROR: Failed to connect to database"}]

```

As we can see in the above picture, logs are being filtered as we generate them.

Conclusion: In conclusion, this case study showcases the efficient use of AWS Lambda to automate log generation and processing in CloudWatch Logs. By generating random logs, filtering for specific error messages like "ERROR," and storing the filtered logs in Amazon S3, the Lambda function enhances real-time operational monitoring. The integration with S3 ensures scalable log storage and easy access for further analysis. This serverless solution eliminates the need for manual log management, optimizes resource usage, and minimizes operational costs. Overall, the automated system improves application reliability and facilitates faster incident resolution, enhancing performance monitoring in real time.

