

EXPORT THE CLOUDWATCH LOGS TO S3 BY USING CLOUDWATCH EVENT RULES AND LAMBDA FUNCTION

Key Uses of Exporting CloudWatch Logs to S3 via Lambda:

1. Long-Term Storage and Archiving:

- CloudWatch Logs can be expensive to retain over a long period due to the costs associated with storage. By exporting logs to Amazon S3, you can store the logs at a lower cost, especially when archived in S3 Glacier for infrequent access. S3 provides virtually unlimited storage and various cost-effective storage classes, which makes it ideal for long-term archiving.

2. Data Backup and Recovery:

- Exporting CloudWatch logs to S3 ensures that log data is backed up in a durable, cost-effective manner. In case of accidental deletion or loss of logs from CloudWatch, the S3 copies remain available for recovery.

3. Centralized Log Management:

- If your organization collects logs from various AWS services and accounts, exporting logs from CloudWatch to S3 can help centralize log management in a single location. This makes it easier to analyze logs across services, track application performance, and troubleshoot issues.

4. Log Analysis with External Tools:

- Once the logs are in S3, you can use third-party log analysis tools or AWS services like **AWS Athena** or **AWS Redshift** to perform deeper analysis on the logs, create reports, and derive insights that can help with monitoring, security auditing, or debugging.

5. Data Processing and Automation:

- AWS Lambda allows you to automate the export process, making it possible to trigger the log export on a schedule (e.g., every day or week). It can also be used to filter or transform logs before they are exported to S3. This can help with data normalization or conversion for easier analysis later.

6. Compliance and Auditing:

- Some regulatory or compliance requirements mandate that log data be retained for a certain period. By exporting CloudWatch logs to S3, you can ensure that you are meeting the retention requirements. Additionally, the logs in S3 can be encrypted and access-controlled, which helps ensure data security and integrity.

7. Cost Control:

- Exporting logs to S3 can help reduce the cost of CloudWatch logs, especially as log volumes grow. CloudWatch charges for log data retention, and moving older logs to S3 allows you to manage costs more effectively by leveraging cheaper storage options in S3.

Typical Workflow Using Lambda:

1. CloudWatch Logs Trigger Lambda:

- A Lambda function can be triggered on a scheduled basis (using Amazon CloudWatch Events or EventBridge) or when new log data is created in CloudWatch Logs.

2. Lambda Retrieves Logs:

- The Lambda function queries the CloudWatch Logs and retrieves log streams or log groups.

3. Process and Format Logs (Optional):

- The Lambda function can process or filter the logs to remove unnecessary information, format them, or perform data transformations.

4. Store Logs in S3:

- The Lambda function then uploads the processed or raw logs to an S3 bucket. You can organize the logs into folders by date, log group, or any other classification that makes sense for your use case.

5. Monitor and Automate:

- The Lambda function can be scheduled to run periodically (using CloudWatch Events) or as an event-driven function triggered by new CloudWatch log entries.

- Add this policy in s3 bucket bucket policy then change the your bucket name and log group names and regions.this policy is mandatory for cloudwatch logs exporting

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "logs.ap-south-1.amazonaws.com"
      },
      "Action": "s3:GetBucketAcl",
      "Resource": "arn:aws:s3:::asdfghjknk"
    },
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "logs.ap-south-1.amazonaws.com"
      },
      "Action": "s3:PutObject",
```

```
"Resource": "arn:aws:s3:::asdfghjknk/*",
```

```
"Condition": {
```

```
  "StringEquals": {
```

```
    "aws:SourceAccount": "908027386826",
```

```
    "s3:x-amz-acl": "bucket-owner-full-control"
```

```
  }
```

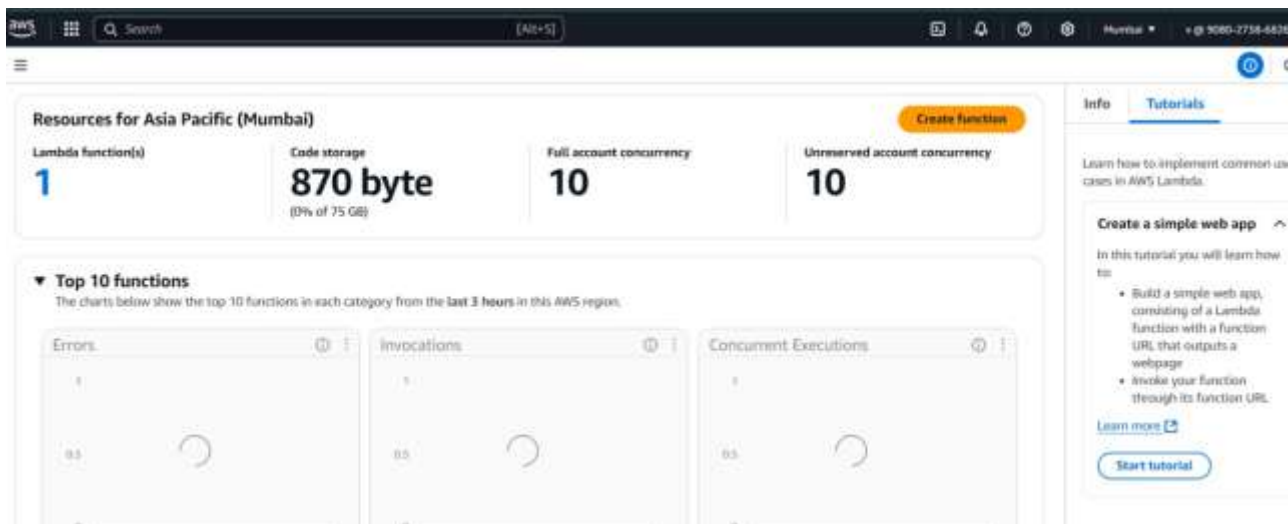
```
 }
```

```
 }
```

```
 ]
```

```
 }
```

- OPEN Lambda in in aws console
- Click on create function



- Enter function name
- Select run time python

[Lambda](#) > [Functions](#) > Create function

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 default role later when adding triggers.

- Change the execution role before you create role for lambda add the admin access or cloudwatch and s3

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 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

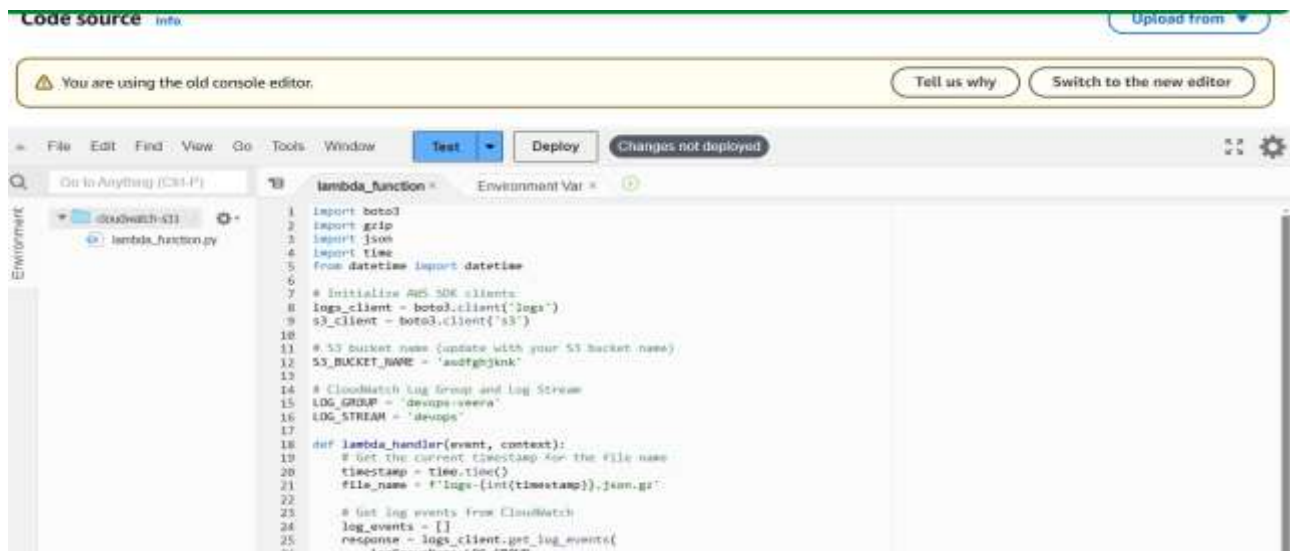
Existing role
 Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.

[View the lambda-admin 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.

[Cancel](#) [Create function](#)

- Enter the below script in lambda function file
- Change the bucket name and log group details
- After script just click on deploy



- and click on test it will ask event configuration
- Enter name and save it

import boto3

import gzip

import json

import time

from datetime import datetime

Initialize AWS SDK clients

logs_client = boto3.client('logs')

s3_client = boto3.client('s3')

S3 bucket name (update with your S3 bucket name)

S3_BUCKET_NAME = 'asdfghjknk'

CloudWatch Log Group and Log Stream

LOG_GROUP = 'devops-veera'

LOG_STREAM = 'devops'

def lambda_handler(event, context):

Get the current timestamp for the file name

timestamp = time.time()

file_name = f'logs-{int(timestamp)}.json.gz'

```

\

# Get log events from CloudWatch
log_events = []

response = logs_client.get_log_events(
    logGroupName=LOG_GROUP,
    logStreamName=LOG_STREAM,
    startTime=int(time.time() - 86400) * 1000, # Get logs from the last 24 hours (in milliseconds)
    endTime=int(time.time()) * 1000,
    limit=10000 # Adjust based on the number of logs you want to fetch
)

# Loop through the response to collect log events
for event in response['events']:
    log_events.append(event)

# Compress logs to gzip format
log_data = json.dumps(log_events, default=str)
compressed_log_data = gzip.compress(log_data.encode('utf-8'))

# Upload to S3
s3_client.put_object(
    Bucket=S3_BUCKET_NAME,
    Key=f'cloudwatch-logs/{file_name}',
    Body=compressed_log_data,
    ContentType='application/gzip'
)

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

```

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 vs. **test**

vs

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

hello-world

Event JSON [Format JSON](#)

```

1 {
2   "key1": "value1",
3   "key2": "value2",
4   "key3": "value3"
}

```

[Cancel](#) [Invoke](#) [Save](#)

- Open configuration
- Select general configuration and edit it

Code Test Monitor **Configuration** Aliases Versions

General configuration [info](#) [Edit](#)

Description	Memory 128 MB	Ephemeral storage 512 MB
Timeout 0 min 3 sec	SnapStart info None	

General configuration

Triggers

Permissions

Destinations

Function URL

Environment variables

Tags

Create a

In this tut
for:

- Buil
con
fun
URI
wel
• Inv
thin

[Learn mo](#)

[Start](#)

- Change the timeout 10 min

[Lambda](#) > [Functions](#) > [cloudwatch-s3](#) > Edit basic settings

Edit basic settings

Basic settings [info](#)

Description - optional

Memory [info](#)
Your function is allocated CPU proportional to the memory configured.
128 MB
Set memory to between 128 MB and 10240 MB.

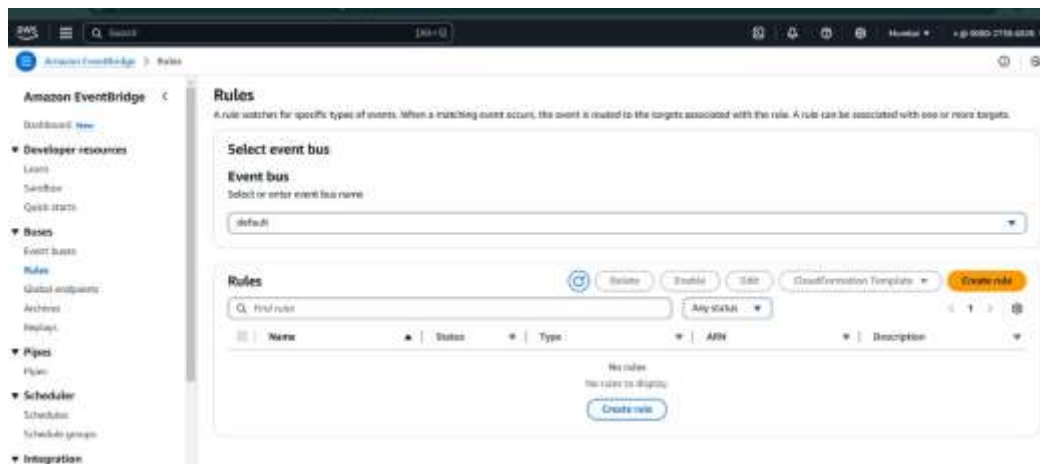
Ephemeral storage [info](#)
You can configure up to 10 GB of ephemeral storage (/tmp) for your function. [View pricing](#)
512 MB
Set ephemeral storage (/tmp) to between 512 MB and 10240 MB.

SnapStart [info](#)
Reduce startup time by having Lambda cache a snapshot of your function after the function has initialized. To evaluate whether your function code is resilient to snapshot operations, review the [SnapStart compatibility considerations](#)
None

Supported runtimes: Java 11, Java 17, Java 21.

Timeout
8 min 20 sec

- Open the cloudwatch
- Click on rules
- Click on create rule



- Enter rule name
- Rule type select schedule
- Click on **continue to create rule**

Rule detail

Name

Maximum of 64 characters, consisting of numbers, lower/upper case letters, ., -, and _.

Description - optional

Event bus [Info](#)

Select the event bus this rule applies to, either the default event bus or a custom or partner event bus.

☒ Enable the rule on the selected event bus

Rule type [Info](#)

☐ Rule with an event pattern

A rule that runs when an event matches the defined event pattern. EventBridge sends the event to the specified target.

☒ **Schedule**

A rule that runs on a schedule

EventBridge Scheduler - A new AWS scheduling capability! [New](#)

A new EventBridge scheduling functionality that provides one-time and recurring scheduling functionality independent of Event buses and rules. You can create a schedule to invoke targets such as a Lambda function.

[Learn More](#)

[Continue to create rule](#)

[Cancel](#)
[Continue in EventBridge Scheduler](#)

[Learn more](#)

Continue to create rule

- Give the corn expression based on utc timings so pass corn expression In utc time
- * means everyday, every month etc

Schedule pattern

Schedule pattern

Choose the schedule type that best meets your needs.

☒ A fine-grained schedule that runs at a specific time, such as 8:00 a.m. PST on the first Monday of every month.

☐ A schedule that runs at a regular rate, such as every 10 minutes.

Cron expression [Info](#)

Define the cron expression for the schedule

 cron ()

Minutes

Hours

Day of month

Month

Day of week

Year

Next 10 trigger date(s)

Local time zone 

Sun, Dec 8, 2024, 11:30 PM GMT+5:30
Mon, Dec 9, 2024, 11:30 PM GMT+5:30
Tue, Dec 10, 2024, 11:30 PM GMT+5:30
Wed, Dec 11, 2024, 11:30 PM GMT+5:30
Thu, Dec 12, 2024, 11:30 PM GMT+5:30
Fri, Dec 13, 2024, 11:30 PM GMT+5:30
Sat, Dec 14, 2024, 11:30 PM GMT+5:30
Sun, Dec 15, 2024, 11:30 PM GMT+5:30
Mon, Dec 16, 2024, 11:30 PM GMT+5:30
Tue, Dec 17, 2024, 11:30 PM GMT+5:30

[Cancel](#)

[Previous](#)

[Next](#)

- Select aws service
- Select target as lambda function
- Select your function

Select target(s)

Permissions

Note: When using the EventBridge console, EventBridge will automatically configure the proper permissions for the selected targets. If you're using the AWS CLI, SDK, or CloudFormation, you'll need to configure the proper permissions.

Target 1

Target types

Select an EventBridge event bus, EventBridge API destination (SaaS partner), or another AWS service as a target.

- ☐ EventBridge event bus
☐ EventBridge API destination
☒ AWS service

Select a target [Info](#)

Select target(s) to invoke when an event matches your event pattern or when schedule is triggered (limit of 5 targets per rule)



Function

► [Configure version/alias](#)

► [Additional settings](#)

- Click on next

Target 1

Target types
Select an EventBridge event bus, EventBridge API destination (SaaS partner), or another AWS service as a target.

☐ EventBridge event bus
☐ EventBridge API destination
☒ AWS service

Select a target [Info](#)
Select target(s) to invoke when an event matches your event pattern or when schedule is triggered (limit of 5 targets per rule)

Lambda function ▼

Function
cloudwatch-s3 ▼ ↻

► **Configure version/alias**

► **Additional settings**

[Add another target](#) [Cancel](#) [Skip to Review and create](#) [Previous](#) [Next](#)

- Click on next

Tags

A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

No tags associated with the resource.

[Add new tag](#)

You can add 50 more tags

[Cancel](#) [Previous](#) [Next](#)

- Review the configuration

Review and create

Step 1: Define rule detail

[Edit](#)

Define rule detail

Rule name lambda-s3	Status Enabled	Event bus default
Description	Rule type Scheduled rule	

Step 2: Build schedule

[Edit](#)

Event schedule [Info](#)

Cron expression

00 18 * * ? *

Next 10 trigger date(s)

Local time zone

- Click on create rule

Additional parameters: --

Dead-letter queue (DLQ): -

Step 4: Configure tag(s)

[Edit](#)

Tags (0)

A tag is a label that you assign to an AWS resource. Each tag consists of a key and an optional value. You can use tags to search and filter your resources or track your AWS costs.

Key	Value
-----	-------

No tags associated with this resource.

[Cancel](#)[Previous](#)[Create rule](#)

- Rule is created
- Attach this role to lambda triggers

Rules (1)

Delete

Enable

Edit

CloudFormation Template

Create rule

Find rules

Any status

< 1 >

<input type="checkbox"/>	Name	Status	Type	ARN	Description
<input type="checkbox"/>	lambda-s3	Enabled	Scheduled Standard	arn:aws:events:ap-south-1:908027386826:rule/lambda-s3	

- Click on add trigger

▼ Function overview [Info](#)

[Export](#)**Diagram**

Template

**cloudwatch-s3**

Layers

(0)

[+ Add trigger](#)[+ Add destination](#)

- Select source as event bridge

Trigger configuration [Info](#)

Select a source

event



Batch/bulk data processing

**EventBridge (CloudWatch Events)**

aws asynchronous schedule management-tools



SQS

EventBridge (CloudWatch Events)

- In existing rules select your rule
- Click on add

Add trigger

Trigger configuration [Info](#)

**EventBridge (CloudWatch Events)**

aws asynchronous schedule management-tools

Rule

Pick an existing rule, or create a new one.

☐ Create a new rule☒ Existing rules

Existing rules

Pick an existing rule

aws:events:ap-south-1:908027386826:rule/lambda-s3



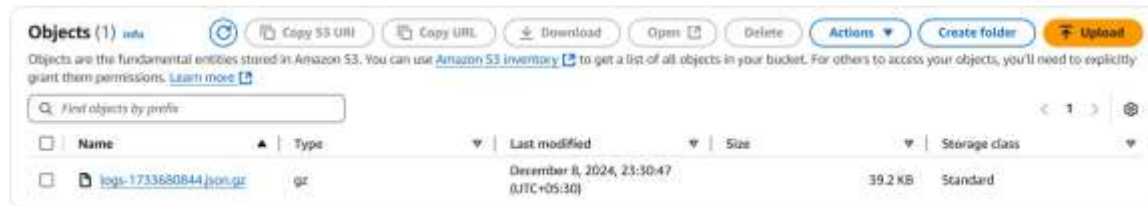
Schedule expression

cron(00 18 * * ? *)

Lambda will add the necessary permissions for Amazon EventBridge (CloudWatch Events) to invoke your Lambda function from this trigger. [Learn more](#) about the Lambda permissions model.

[Cancel](#)[Add](#)

- Once your schedule is reached lambda will trigger and run the lambda function
- The cloud watch logs is exported into s3 bucket successfully



- After running lambda function run based on event rules we will see the status of the function by using sns

Step 1: Create an SNS Topic

1. Log in to the AWS Management Console.
2. Navigate to **SNS** (under "Services").
3. In the left-hand panel, click on **Topics**.
4. Click on **Create topic**.
 - Choose the **Type** of topic:
 - **Standard:** Best for most use cases, including fan-out messaging.
 - **FIFO:** If you need ordered message delivery (optional).
 - Enter a **Name** for your topic.
 - Leave other settings at their default, or adjust them as needed (e.g., enabling delivery retry policies).
5. Click **Create topic** to finish.

Step 2: Create a Lambda Function

1. Navigate to the **Lambda** service in the AWS Management Console.
2. Click on **Create function**.
3. Choose **Author from scratch**.
 - Enter a **Function name**.
 - Choose a **Runtime** (e.g., Python, Node.js).
4. For **Permissions**, you can create a new role with basic Lambda permissions or use an existing role.
5. Click **Create function** to finish.

Step 3: Add an SNS Trigger to Your Lambda Function

1. After your Lambda function is created, in the Lambda function's page, under the **Function overview** section, click on **Add trigger**.
2. From the list of triggers, select **SNS**.
3. In the **SNS topic** field, choose the SNS topic you created earlier.
4. If prompted, give Lambda permission to access SNS by clicking the **Add permission** button, which will automatically add the necessary permissions for Lambda to be invoked by SNS.
5. Click **Add** to finish.

Step 4: Configure Permissions for Lambda to Access SNS

For the Lambda function to be triggered by SNS, Lambda needs permissions to read from the SNS topic. AWS does this by automatically updating the Lambda execution role when you add SNS as a trigger (in step 3). However, if you're doing it manually or need to ensure the correct permissions, follow these steps:

1. In the **IAM Console**, locate the **Lambda execution role**.
2. Attach the **AWSLambdaBasicExecutionRole** and **AmazonSNSReadOnlyAccess** managed policies, or create a custom policy to allow SNS message consumption.
3. Example custom policy for SNS access:

json

Copy code

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "sns:Publish",
      "Resource": "*"
    }
  ]
}
```

Step 5: Test the Configuration

1. **Publish a message to the SNS topic:**
 - Go back to **SNS** in the AWS Console.
 - Click on the **Topic** you created.
 - Click **Publish message**.
 - Enter a sample message in the **Message body**.

- Click **Publish**.

2. Verify Lambda invocation:

- Go to your **Lambda function's** page.
- Check the **CloudWatch Logs** for the Lambda function to confirm it was triggered by the SNS message.

Step 6: Optional - Confirm Lambda Execution Results

You can use **CloudWatch Logs** to monitor how your Lambda function processes the SNS notifications. You can also log custom details within your Lambda function to ensure it's performing as expected.

Example: Lambda Code to Handle SNS Notifications (Python)

Here's an example Lambda function in Python that logs the received SNS message:

python

Copy code

```
import json
```

```
def lambda_handler(event, context):
```

```
    # Log the SNS event received
```

```
    print("Received event: " + json.dumps(event, indent=2))
```

```
    # Example of extracting the message from the event
```

```
    message = event["Records"][0]["Sns"]["Message"]
```

```
    print(f"Message received: {message}")
```

```
    # You can add further processing logic here, depending on your use case
```

```
    return {
```

```
        'statusCode': 200,
```

```
        'body': json.dumps('SNS message processed successfully')
```

```
    }
```

Summary of Steps:

1. Create SNS topic.
2. Create Lambda function.
3. Add SNS trigger to Lambda (automatically adds permissions).

4. Publish a message to SNS and verify Lambda receives it.

With this setup, whenever a message is published to your SNS topic, it will automatically trigger your Lambda function.

You said:

how to add sns in asynchronus in lambda

ChatGPT said:

ChatGPT

To add SNS as an **asynchronous invocation** trigger for your AWS Lambda function, you can configure SNS to send messages to Lambda in a way that Lambda executes asynchronously. This means that once SNS sends a message to Lambda, Lambda queues the function for processing, and the processing will happen independently of the SNS event (i.e., SNS does not wait for the Lambda function to complete before returning).

AWS Lambda supports asynchronous invocations by default for certain events, including SNS. When SNS sends a message to Lambda, it will invoke the function asynchronously.

Here's how you can configure an SNS topic to invoke a Lambda function asynchronously:

Step-by-Step Guide

1. Create an SNS Topic

- Go to the **SNS** section of the AWS Management Console.
- Click on **Create topic** and follow the steps to create an SNS topic (choose **Standard** or **FIFO**, depending on your use case).
- After the topic is created, make a note of the **ARN** (Amazon Resource Name) for the topic.

2. Create a Lambda Function

- Navigate to the **Lambda** section in the AWS Management Console.
- Click on **Create function** and choose the **Author from scratch** option.
- Give your function a name (e.g., `MyAsyncLambdaFunction`).
- Choose your preferred runtime (e.g., Python, Node.js).
- In the **Permissions** section, create or use an existing IAM role with at least the following basic permissions:
 - `AWSLambdaBasicExecutionRole` for logging to CloudWatch.
 - `AmazonSNSFullAccess` or a custom policy to allow Lambda to read messages from SNS.

Here's an example of a simple Lambda function in Python that processes SNS notifications asynchronously:

python

Copy code


```
import json
```

```
def lambda_handler(event, context):
```

```
    # Log the SNS event to CloudWatch Logs
```

```
    print("Received event:", json.dumps(event, indent=2))
```

```
    # Extract the message from the SNS event
```

```
    message = event['Records'][0]['Sns']['Message']
```

```
    # Process the message (you can add your logic here)
```

```
    print(f"Processing message: {message}")
```

```
    # Return a response (not required for asynchronous invocations)
```

```
    return {
```

```
        'statusCode': 200,
```

```
        'body': json.dumps('SNS message processed asynchronously')
```

```
    }
```

3. Add SNS as an Event Source for Lambda (Asynchronous Invocation)

To configure SNS to trigger your Lambda function asynchronously, follow these steps:

1. Go to the Lambda Console:

- Open your Lambda function in the AWS Management Console.

2. Add SNS Trigger:

- In the **Function overview** section, click on **Add trigger**.
- Choose **SNS** as the trigger source.
- In the **SNS topic** dropdown, select the SNS topic you created earlier.
- Check the box **Enable trigger** to activate the trigger immediately.

3. Lambda Execution Role Permissions:

- When you add SNS as a trigger, AWS will automatically update your Lambda's execution role with permissions to access SNS. If needed, you can manually verify that the Lambda role has the required SNS permissions, which are typically:
 - `sns:Subscribe` - to subscribe Lambda to the SNS topic.
 - `sns:Receive` - to allow Lambda to receive messages from SNS.

Example inline policy that grants SNS permissions:

json

Copy code

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "sns:Subscribe",
        "sns:Receive"
      ],
      "Resource": "arn:aws:sns:region:account-id:topic-name"
    }
  ]
}
```

4. Save the Trigger:

- Click on **Add** to save the SNS trigger configuration.

4. Asynchronous Invocation Behavior

By default, SNS invokes Lambda functions asynchronously. This means that:

- SNS sends the message to the Lambda function without waiting for the function's response.
- If the Lambda function is successfully invoked, SNS considers the operation complete (even if the Lambda function is still processing).
- If the Lambda function fails (e.g., due to an error), SNS retries the invocation twice (with delays between retries), following the default retry behavior for asynchronous invocations.

5. Monitor and Handle Failures (Optional)

Asynchronous invocations come with automatic retries, but there might be cases where you want to explicitly handle failures. You can configure a **Dead Letter Queue (DLQ)** or **Amazon EventBridge** for error handling and logging:

- **Dead Letter Queue (DLQ):**
 - Configure an SNS topic or SQS queue as the dead letter queue (DLQ) for your Lambda function. This ensures that failed invocations are captured for later analysis and reprocessing.
 - In the Lambda Console, under **Destinations**, you can configure an SQS queue or SNS topic for DLQ.

- **EventBridge:**

- You can also route failed Lambda invocations to EventBridge to track and manage errors.

Summary of Key Points:

1. **Asynchronous SNS Invocation:** SNS invokes Lambda functions asynchronously by default, meaning it doesn't wait for the Lambda function to finish processing before it returns.
2. **Configure SNS Trigger:** To configure SNS as an event source for your Lambda function, go to the Lambda function console and add an SNS trigger.
3. **Lambda Execution Role:** Ensure that your Lambda function has the correct permissions to allow it to read messages from the SNS topic.
4. **Retry and Failure Handling:** SNS will retry failed Lambda invocations twice. You can configure a DLQ for capturing failed invocations or use EventBridge for advanced failure handling.

This setup allows you to process messages from SNS without blocking the SNS message delivery process, which is useful for high-volume or long-running processing tasks.

You said:

how to crete sns to gmail

ChatGPT said:

ChatGPT

To send notifications from **Amazon SNS (Simple Notification Service)** to a **Gmail address**, you can set up an **SNS subscription** with the **Email** protocol. Here's a step-by-step guide on how to do it:

Steps to Set Up SNS to Send Emails to Gmail

1. Create an SNS Topic

- **Log in to AWS Management Console.**
- Go to the **SNS** service (Search for SNS in the services search bar).
- In the left sidebar, click on **Topics**.
- Click on **Create topic**.
 - **Type:** Select **Standard** (or FIFO if you need ordered delivery).
 - **Name:** Give your topic a name (e.g., MyTopic).
 - Leave other settings at their default values or adjust as needed.
- Click **Create topic** to finish creating the SNS topic.

2. Create an SNS Subscription for Email (Gmail)

- Once your topic is created, you'll be directed to the topic's page.
- Click on the **Create subscription** button.
- **Protocol:** Choose **Email** from the dropdown.

- **Endpoint:** Enter your Gmail address (e.g., your-email@gmail.com).
- Click **Create subscription**.

3. Confirm the Subscription

- After creating the subscription, SNS will send a confirmation email to the Gmail address you specified.
- Go to your **Gmail** inbox and open the email from Amazon SNS.
- Click on the **Confirm subscription** link in the email to confirm the subscription.

Once you confirm the subscription, your Gmail address is now subscribed to the SNS topic and will receive notifications when a message is published to this topic.

- Add the sns in lambda trdestination
- Click on destination



- Select sns topic before that you need to create a sns topic

Add destination

Destination configuration
[Info](#)

Configure a destination to receive invocation records. Lambda can send records when your function is invoked asynchronously, or when your function processes records from an event source mapping.

Source
Choose the invocation type that Lambda sends records for:

☒ Asynchronous invocation
☐ Event source mapping invocation

Condition
Choose whether to send invocation records for event processing failures or for successful invocations.

☒ On failure
☐ On success

Destination type
Choose the destination type that Lambda sends invocation records to:

SNS topic

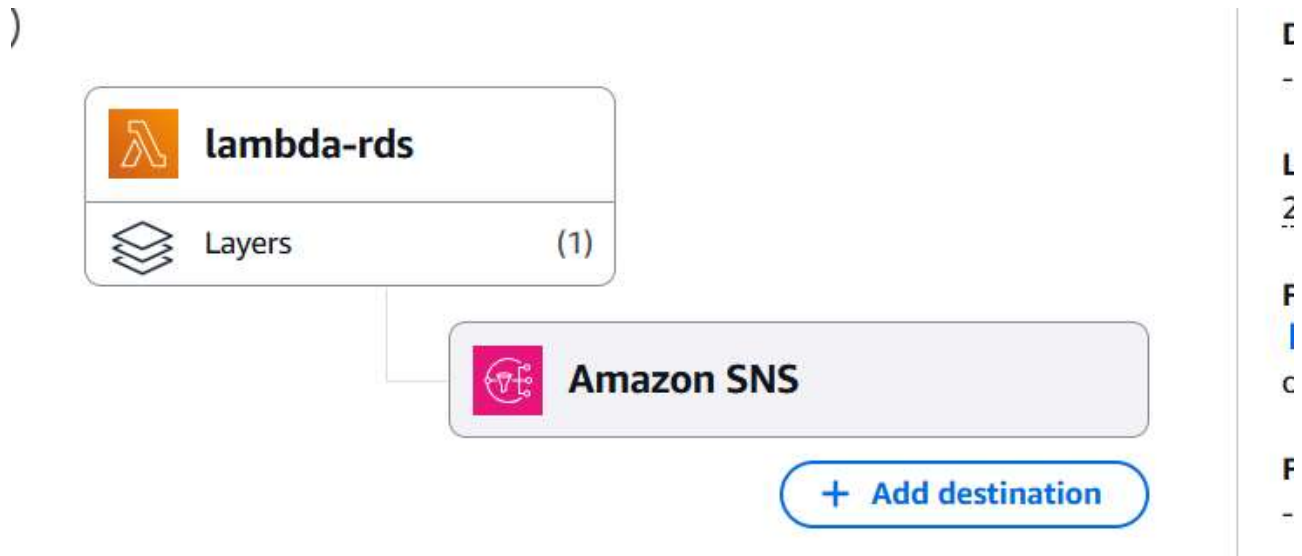
Destination
Choose the ARN of the destination, or enter the ARN manually.

Use: "arn:aws:sns:ap-south-1:908027386826:mytopic"

mytopic

Permissions

- Sns topic attached to lambda function
- Once function will run you will get the notification through sns



■ Lambda function explanation

1. Importing Required Libraries

- **boto3**: AWS SDK for Python, used to interact with AWS services like CloudWatch Logs and S3.
- **gzip**: A module to compress data into the gzip format.
- **json**: Used to convert Python objects (like lists and dictionaries) to JSON format.
- **time**: Used to get the current time and manipulate timestamps.
- **datetime**: Although imported, it's not used in this code but could be useful for handling time in other formats.

2. Initializing AWS SDK Clients

- **logs_client = boto3.client('logs')**: Initializes the AWS CloudWatch Logs client to interact with CloudWatch Logs.
- **s3_client = boto3.client('s3')**: Initializes the AWS S3 client to interact with Amazon S3 for storing the log data.

3. Defining Variables

- **S3_BUCKET_NAME = 'asdfghjknk'**: Specifies the name of the S3 bucket where logs will be uploaded. Replace this with the actual bucket name.
- **LOG_GROUP = 'devops-veera'**: The CloudWatch Logs group from which the logs will be fetched. Replace with the actual log group name.
- **LOG_STREAM = 'devops'**: The specific log stream within the log group that contains the logs to be exported. Replace with the actual stream name.

4. Lambda Handler Function: lambda_handler

This is the main function that will be executed when the Lambda function is triggered. It takes two parameters:

- **event:** Contains details about the event that triggered the Lambda function.
- **context:** Provides runtime information about the Lambda execution environment.

Inside the handler:

a. Generate Timestamped File Name

python

Copy code

```
timestamp = time.time()
file_name = f'logs-{int(timestamp)}.json.gz'
```

- `time.time()` returns the current time as a float in seconds since the epoch (January 1, 1970).
- `file_name` is created using this timestamp, ensuring that the filename is unique. The file will be named `logs-<timestamp>.json.gz`.

b. Get Log Events from CloudWatch Logs

python

Copy code

```
log_events = []
response = logs_client.get_log_events(
    logGroupName=LOG_GROUP,
    logStreamName=LOG_STREAM,
    startTime=int(time.time() - 86400) * 1000, # Get logs from the last 24 hours (in milliseconds)
    endTime=int(time.time()) * 1000,
    limit=10000 # Adjust based on the number of logs you want to fetch
)
```

- **logs_client.get_log_events:** Fetches log events from CloudWatch for the specified log group and stream.
 - `logGroupName` and `logStreamName`: Specify the CloudWatch Log Group and Log Stream to retrieve logs from.
 - **startTime:** The start time (in milliseconds) is set to 24 hours ago (86400 seconds) from the current time.
 - **endTime:** The current time is set as the end time.
 - **limit:** The maximum number of log events to retrieve. In this case, it's set to 10,000 logs, but you can adjust it based on your needs.

c. Store Retrieved Log Events

python

Copy code

```
for event in response['events']:
```

```
    log_events.append(event)
```

- The `get_log_events` API response contains log events. These are extracted and appended to the `log_events` list.

d. Compress Logs into GZIP Format

python

Copy code

```
log_data = json.dumps(log_events, default=str)
```

```
compressed_log_data = gzip.compress(log_data.encode('utf-8'))
```

- **`json.dumps(log_events)`**: Converts the list of log events into a JSON-formatted string.
- **`gzip.compress()`**: Compresses the log data into the gzip format, ensuring efficient storage and transfer to S3.

e. Upload Logs to S3

python

Copy code

```
s3_client.put_object(
```

```
    Bucket=S3_BUCKET_NAME,
```

```
    Key=f'cloudwatch-logs/{file_name}',
```

```
    Body=compressed_log_data,
```

```
    ContentType='application/gzip'
```

```
)
```

- **`s3_client.put_object`**: Uploads the compressed log file to the specified S3 bucket.
 - **Bucket**: The S3 bucket name where the logs will be stored.
 - **Key**: The path and name of the file in S3 (e.g., `cloudwatch-logs/logs-<timestamp>.json.gz`).
 - **Body**: The body of the request, which is the compressed log data.
 - **ContentType**: Specifies the content type as `'application/gzip'` to indicate that the file is in gzip format.

5. Return Response

python

Copy code

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

- Returns a success response with a status code 200 and a body containing a success message in JSON format.

Summary

This Lambda function performs the following actions:

1. Retrieves log events from a specified CloudWatch Log Group and Log Stream from the past 24 hours.
2. Compresses the log data into gzip format.
3. Uploads the compressed log data to an S3 bucket.
4. Returns a success message.

Thank you