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.
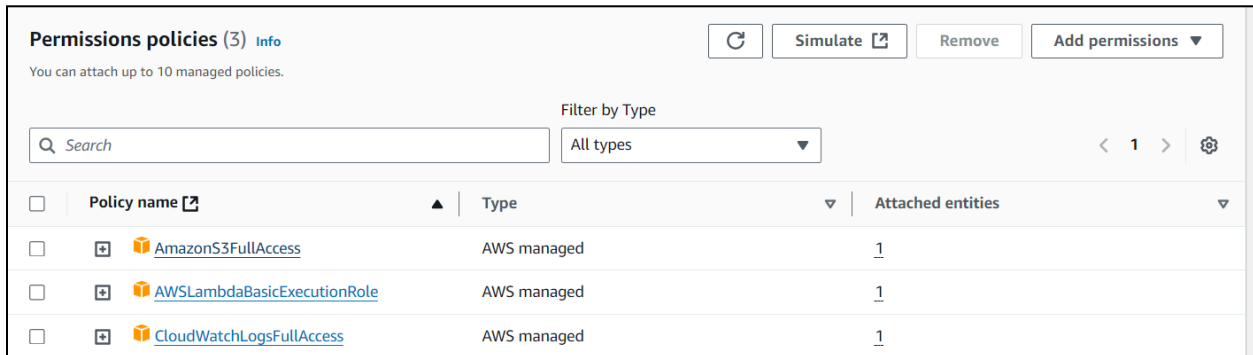
Amazon S3
Amazon S3 (Simple Storage Service) is a scalable object storage service designed for a wide range of use cases, including data storage, backup and restore, archiving, and big data analytics. S3 provides a secure and highly available environment to store any amount of data from anywhere. You 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.
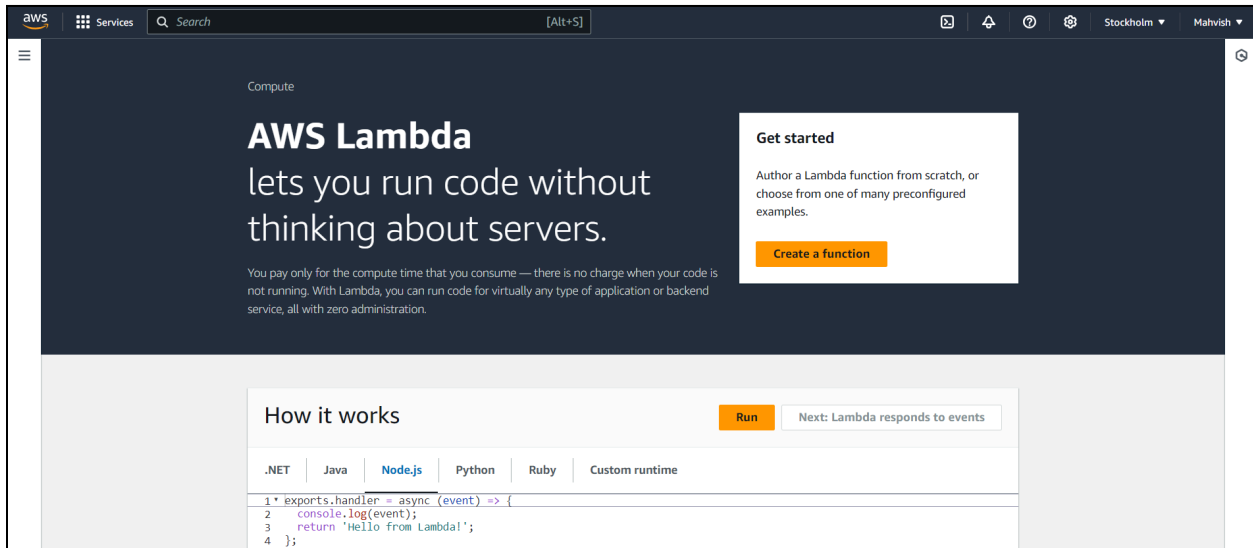
Step-by-Step Explanation

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.

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

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

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

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

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
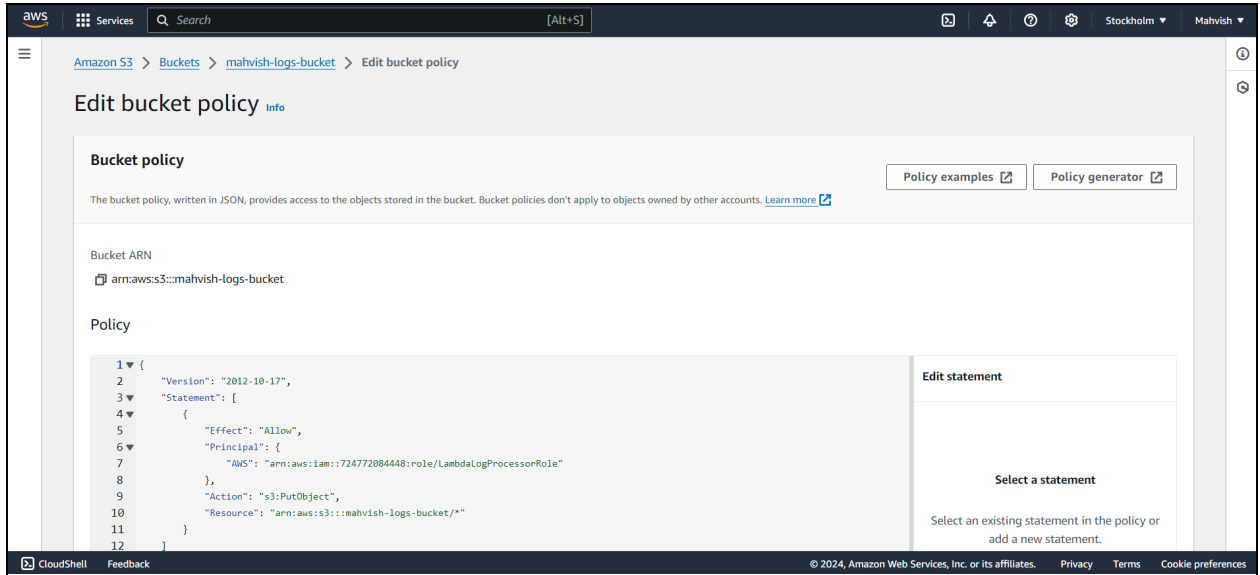
Step 7: Create a cloudswatch log group for saving the logs created by our lambda function
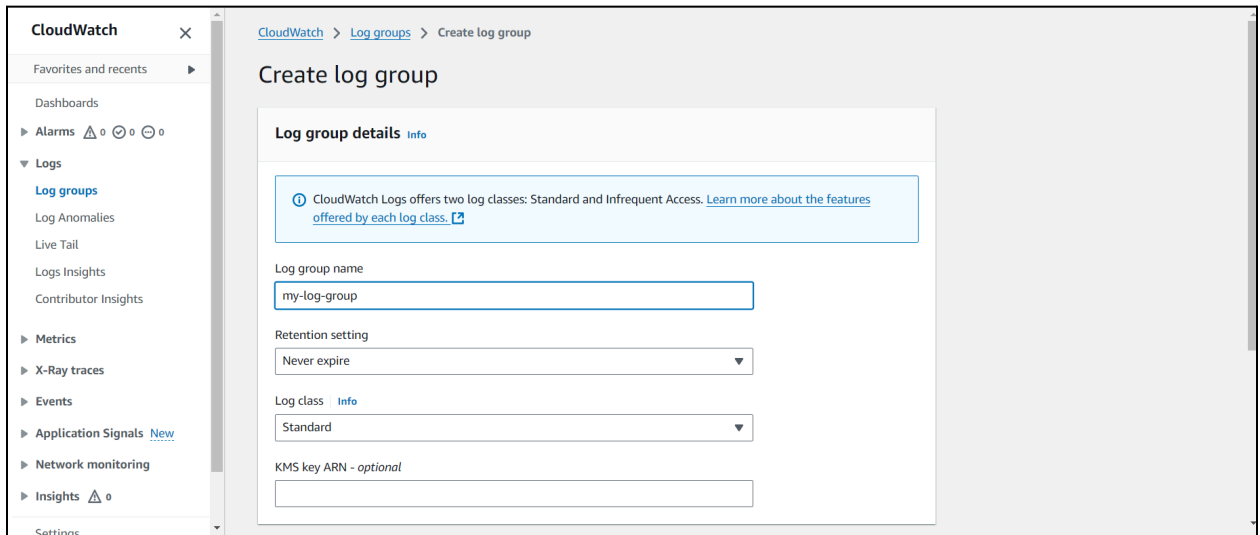
Step 8: Now, go back to the lambda function and add the cloudwatch group in triggers
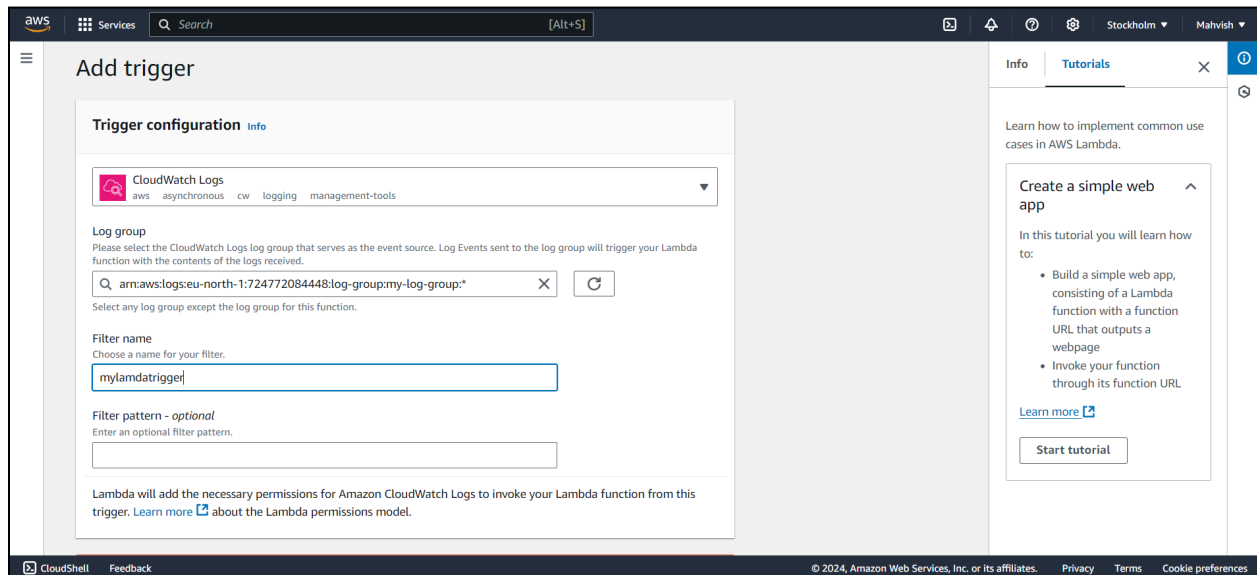
Select the log group created in the previous step

Step 9: Add the code for lambda function

Function code:

```
import boto3
import json

s3_client = boto3.client('s3')

def lambda_handler(event, context):
    try:
        log_events = event['logEvents']
        print(f"Received {len(log_events)} log events")

        # Process log event data
        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:
            s3_client.put_object(
                Bucket='mahvish-logs-bucket',
                Key='filtered_logs.json',
                Body=json.dumps(filtered_logs)
            )
```
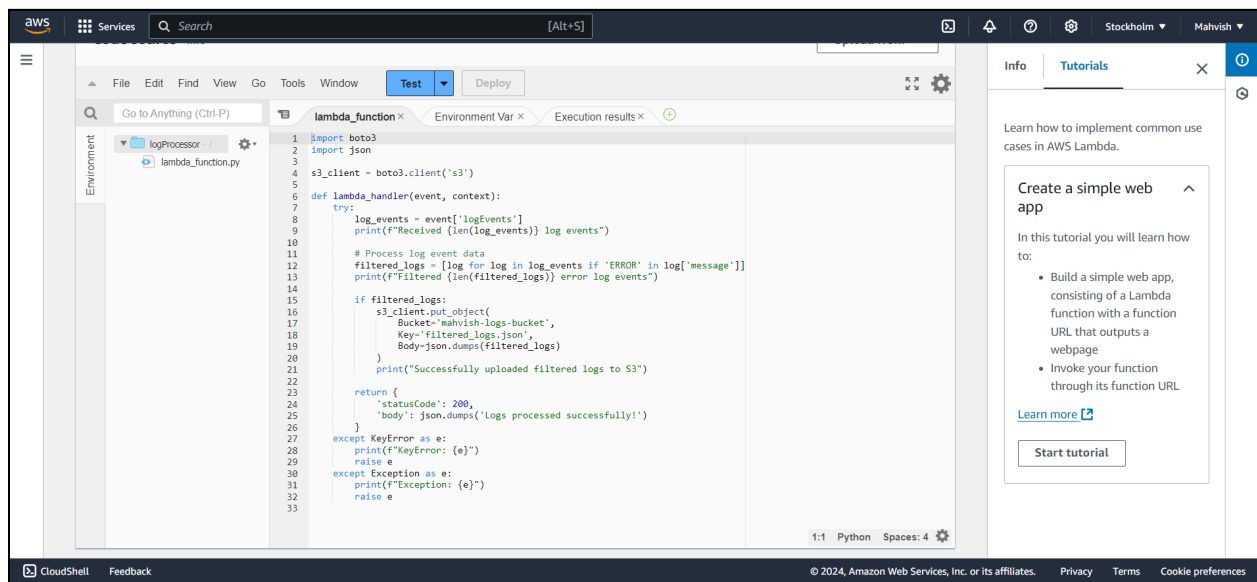
```python
        print("Successfully uploaded filtered logs to S3")

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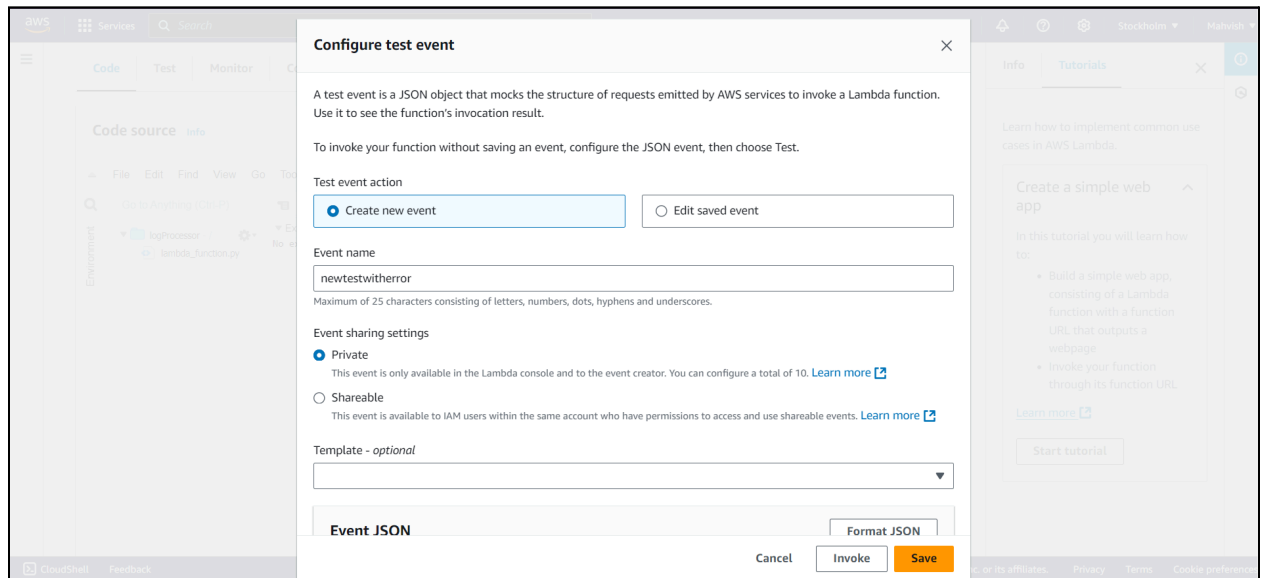
Note: In the above code, add the name of S3 bucket created earlier.

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

Step 10: 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."
    }
  ]
}
```

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

Step 11: 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.

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