Ex.No :1	Setup and Configure EC2 Instance with Security Group Rules and Perform Basic C Programming Operations

AIM:

To install and configure a virtual machine on Amazon EC2, set up a custom security group named ―VPC Lab‖ with rules allowing MySQL (port 3306) from web-sec-grp, allow outbound internet traffic, test HTTP connectivity, and install GCC (GNU Compiler Collection) on a Linux instance to run a basic C program.

ALGORITHM:

1. Login to the AWS Management Console and navigate to the EC2 Dashboard.
2. Launch a new EC2 instance with Amazon Linux or Ubuntu as the base OS.
3. Create a new security group named ―VPC Lab‖.
4. Configure inbound rule to allow MySQL traffic (port 3306) from the ―web-sec-grp‖ security group.
5. Configure outbound rule to allow all traffic to the internet.
6. Launch the instance with an existing or new key pair and connect using SSH.
7. Test inbound HTTP using curl and outbound connectivity using ping or wget.
8. Install GCC (GNU Compiler Collection) using the Linux package manager.
9. Write and compile a basic C program for arithmetic operations.
10. Execute the program to verify successful installation and functionality.

PROGRAM:

```
#include <stdio.h>

int main() {
   printf("Hello, EC2 Instance!\n");

   // Basic operations
   int a = 10, b = 5;

   printf("Addition: %d + %d = %d\n", a, b, a + b);
   printf("Subtraction: %d - %d = %d\n", a, b, a - b);
```

```
    printf("Multiplication: %d * %d = %d\n", a, b, a * b);
    printf("Division: %d / %d = %d\n", a, b, a / b);

    return 0;
}
```

OUTPUT:

```
Hello, EC2 Instance!
Addition: 10 + 5 = 15
Subtraction: 10 - 5 = 5
Multiplication: 10 * 5 = 50
Division: 10 / 5 = 2
```

RESULT:

The virtual machine was successfully created using Amazon EC2. The security group ―VPC Lab‖ was configured to allow MySQL traffic on port 3306 and outbound internet access. Inbound and outbound HTTP traffic was successfully tested. The GCC compiler was installed, and the C program executed successfully, performing basic operations like addition, subtraction, multiplication, and division.

Ex.No :2	Launch an EC2 Instance and Deploy a Hello World Web Application with CloudWatch Monitoring

AIM:

To launch an EC2 instance using Amazon Web Services (AWS), configure it with the necessary settings, deploy a basic ―Hello World‖ web application, and enable **CloudWatch** to monitor logs and system metrics for identifying and resolving issues.

ALGORITHM:

1. Login to the **AWS Management Console** and go to **EC2 Dashboard**.
2. Click **Launch Instance** and choose an **Amazon Machine Image (AMI)** (e.g., Amazon Linux 2).
3. Select an **Instance Type** (e.g., t2.micro – Free Tier eligible).
4. Create or choose a **Key Pair** to access the instance via SSH.
5. Create a new **Security Group** to allow **HTTP (port 80)** and **SSH (port 22)** traffic.
6. Launch the instance and connect to it using an SSH terminal.
7. Install a **web server** (e.g., Apache HTTP server).
8. Create an HTML file named **index.html** with "Hello World" content.
9. Copy the file to the web server's document root directory (e.g., `/var/www/html`).
10. Open the instance's public IP address in a browser to verify the output.
11. Enable **CloudWatch monitoring** to collect and view EC2 logs and metrics.

PROGRAM:

```
 # Step 1: Connect to EC2 Instance
ssh -i "your-key.pem" ec2-user@<your-ec2-public-ip>

# Step 2: Install Apache Web Server
sudo yum update -y
sudo yum install httpd -y

# Step 3: Start the Web Server
sudo systemctl start httpd
sudo systemctl enable httpd
```

```
# Step 4: Create and Deploy Hello World Page
echo "Hello World" | sudo tee /var/www/html/index.html

# Step 5: Verify Web Page
curl http://localhost
```

OUTPUT:

RESULT:

The EC2 instance was successfully launched and configured. The **Apache web server** was installed, and a simple **"Hello World" HTML** page was deployed. The application was successfully accessed over the internet through the instance's public IP. Additionally, **CloudWatch monitoring** was enabled to track logs and system performance for troubleshooting and metrics analysis.

Ex.No :3	Create IAM Users, Groups, and Custom Policies in AWS

AIM:

To create IAM users, groups, and custom policies in AWS. This includes assigning predefined and custom permissions to users and groups to securely manage access to AWS services like EC2, S3, and DynamoDB.

ALGORITHM:

1. **Login** to the AWS Management Console and go to the **IAM (Identity and Access Management)** service.
2. **Create IAM user** named **"user1"** with **programmatic access** enabled.
3. Attach the **AmazonEC2ReadOnlyAccess** policy to **user1**.
4. Create an **IAM group** named **"admin-group"**.
5. Create two IAM users: **"admin1"** and **"admin2"**.
6. Add both users (**admin1** and **admin2**) to the **"admin-group"**.
7. Attach the **AdministratorAccess** policy to **admin-group**.
8. Create a new **IAM policy** named **"custom-policy"** with permissions to **Amazon S3** and **Amazon DynamoDB**.
9. Create an IAM user named **"user2"**.
10. Attach the **"custom-policy"** to **user2**.

PROGRAM:

```
# 1. Create IAM user 'user1' with programmatic access
aws iam create-user --user-name user1
aws iam create-access-key --user-name user1

# 2. Attach AmazonEC2ReadOnlyAccess policy to 'user1'
aws iam attach-user-policy \
  --user-name user1 \
  --policy-arn arn:aws:iam::aws:policy/AmazonEC2ReadOnlyAccess
```

```
# 3. Create IAM group 'admin-group'
aws iam create-group --group-name admin-group

# 4. Create IAM users 'admin1' and 'admin2'
aws iam create-user --user-name admin1
aws iam create-user --user-name admin2

# 5. Add users to 'admin-group'
aws iam add-user-to-group --user-name admin1 --group-name admin-group
aws iam add-user-to-group --user-name admin2 --group-name admin-group

# 6. Attach AdministratorAccess policy to 'admin-group'
aws iam attach-group-policy \
  --group-name admin-group \
  --policy-arn arn:aws:iam::aws:policy/AdministratorAccess

# 7. Create a custom policy (custom-policy) for S3 and DynamoDB
cat > custom-policy.json <<EOF
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "s3:*",
            "dynamodb:*"
         ],
         "Resource": "*"
      }
   ]
}
EOF

aws iam create-policy \
  --policy-name custom-policy \
  --policy-document  file://custom-policy.json

# 8. Create IAM user 'user2'
aws iam create-user --user-name user2

# 9. Attach custom-policy to 'user2'
aws iam attach-user-policy \
  --user-name user2 \
  --policy-arn  arn:aws:iam::<your-account-id>:policy/custom-policy
```

OUTPUT:

Create IAM User "user1"


```
{
    "User": {
        "Path": "/",
        "UserName": "user1",
        "UserId": "AIDAEXAMPLEUSERID1",
        "Arn": "arn:aws:iam::123456789012:user/user1",
        "CreateDate": "2025-07-13T10:15:30+00:00"
    }
}
```


Create Group "admin-group"


```
$ aws iam create-group --group-name admin-group

{
    "Group": {
        "Path": "/",
        "GroupName": "admin-group",
        "GroupId": "AGPAEXAMPLEGROUPID1",
        "Arn": "arn:aws:iam::123456789012:group/admin-group",
        "CreateDate": "2025-07-13T10:20:00+00:00"
    }
}
```

Create Custom Policy

```
$ aws iam create-policy \
--policy-name custom-policy \
--policy-document file://custom-policy.json

{
    "Policy": {
        "PolicyName": "custom-policy",
        "PolicyId": "ANPAEXAMPLEPOLICYID",
        "Arn": "arn:aws:iam::123456789012:policy/custom-policy",
        "Path": "/",
        "DefaultVersionId": "v1",
        "AttachmentCount": 0,
        "IsAttachable": true,
        "CreateDate": "2025-07-13T10:25:10+00:00",
        "UpdateDate": "2025-07-13T10:25:10+00:00"
    }
}
```

RESULT:

IAM users, groups, and custom policies were successfully created and configured in AWS. Access permissions were properly assigned using both predefined and custom policies, ensuring secure and role-based access control to AWS resources.

Ex.No :4	Host a Static Website Using Amazon S3 with Public Access, Bucket Policy, and Versioning

AIM:

To create an S3 bucket with a unique name, host a static website by uploading an HTML file, configure public access and bucket policies, securely share objects using pre- signed URLs, and manage object versioning.

ALGORITHM:

1. Login to the **AWS Console** and go to **Amazon S3**.
2. Click **"Create bucket"** and enter a unique name.
3. Uncheck **"Block all public access"** to allow website hosting.
4. Enable **Static Website Hosting** under bucket properties.
5. Create and upload an `index.html` file with website content.
6. Set object permissions to make it **publicly accessible**.
7. Create and apply a **bucket policy** for public read access.
8. Generate a **pre-signed URL** for securely sharing a specific object.
9. Enable **versioning** on the bucket.
10. Upload an updated `index.html` file to generate a new version.
11. Check and display the **Version IDs** of the object in S3 Console.

PROGRAM:

```
# 1. Create a new S3 bucket (use unique name)
aws s3api create-bucket --bucket my-static-site-2025 --region us-east-1

# 2. Enable static website hosting
aws s3 website s3://my-static-site-2025/ --index-document index.html

# 3. Upload index.html file
aws s3 cp index.html s3://my-static-site-2025/

# 4. Set object ACL to public-read
aws s3api put-object-acl --bucket my-static-site-2025 --key index.html --acl public-read
```

```
# 5. Add a bucket policy to allow public read access
aws s3api put-bucket-policy --bucket my-static-site-2025 --policy '{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "PublicReadGetObject",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "s3:GetObject",
      "Resource":  "arn:aws:s3:::my-static-site-2025/*"
    }
  ]
}'

# 6. Generate a pre-signed URL (valid for 1 hour)
aws s3 presign s3://my-static-site-2025/index.html --expires-in 3600

# 7. Enable versioning on the bucket
aws s3api put-bucket-versioning --bucket my-static-site-2025 --versioning-configuration
Status=Enabled

# 8. Upload updated version of index.html
aws s3 cp index.html s3://my-static-site-2025/

# 9. List object versions
aws s3api list-object-versions --bucket my-static-site-2025
```

OUTPUT:


```
{
    "Versions": [
        {
            "ETag": "\"f9f9c7d3c6e8e25b89798cabc9ceef6a\"",
            "Size": 125,
            "StorageClass": "STANDARD",
            "Key": "index.html",
            "VersionId": "3HL4kqtJlcpXrof3JTFdGYpYz2pZtQas",
            "IsLatest": true
        },
        {
            "ETag": "\"2edfd1e8c1c82bba1359f9cf3c2efc14\"",
            "Size": 112,
            "StorageClass": "STANDARD",
            "Key": "index.html",
            "VersionId": "Aqvs1Imc27NR6jmb4C5L3vHYrL8OcD4F",
            "IsLatest": false
        }
    ]
}
```

RESULT:

An Amazon S3 bucket was successfully created and configured to host a static website. The `index.html` file was made publicly accessible through a bucket policy. A pre-signed URL was generated for secure file sharing. Versioning was enabled, and multiple object versions were listed with their unique Version IDs.

Ex.No :5	Convert Text into an Audiobook using Amazon Polly and Store it in Amazon S3

AIM:

To use **Amazon Polly** to convert a text script or blog post into an audio file by selecting and customizing a voice, adjusting pitch and speed, and saving the audio output into an Amazon S3 bucket.

ALGORITHM:

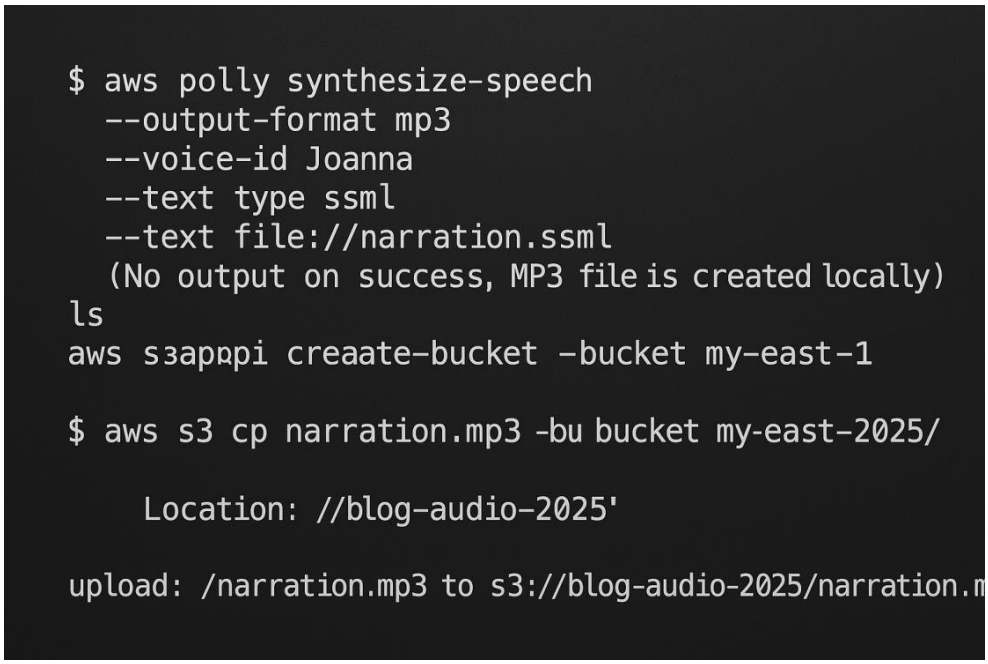
1. Open the **AWS Management Console** and navigate to the **Amazon Polly** service.
2. Choose a voice (e.g., **Joanna**, **Matthew**, or **Amy**) for narration.
3. Write or paste the blog content/script into the Polly editor.
4. Customize voice output by adjusting **pitch** and **speed** using **SSML tags**.
5. Synthesize the speech and generate the audio file in **MP3** format.
6. Download the audio or use the **AWS CLI** to retrieve it programmatically.
7. Create an **S3 bucket** or use an existing one.
8. Upload the audio file to the **S3 bucket**.
9. Set access permissions if needed and confirm successful upload.

PROGRAM:

```
<speak>
  <prosody rate="slow" pitch="+2%">
    Welcome to the world of cloud computing. This audiobook is generated using Amazon Polly.
  </prosody>
</speak>


# Step 1: Create SSML file with custom pitch and speed
echo '<speak><prosody rate="slow" pitch="+2%">Welcome to the world of cloud computing. This
audiobook is generated using Amazon Polly.</prosody></speak>' > narration.ssml

# Step 2: Use Amazon Polly to synthesize speech from the SSML file
aws polly synthesize-speech \
  --output-format mp3 \
  --voice-id Joanna \
  --text-type ssml \
```

```
  --text file://narration.ssml \
  narration.mp3

# Step 3: Create an S3 bucket (use a unique name)
aws s3api create-bucket --bucket blog-audio-2025 --region us-east-1

# Step 4: Upload the audio file to the S3 bucket
aws s3 cp narration.mp3 s3://blog-audio-2025/
```

OUTPUT:



RESULT:

The text blog/script was successfully converted into an audiobook using Amazon Polly. The voice was customized with pitch and speed control using SSML. The resulting MP3 file was saved and uploaded to an Amazon S3 bucket for access and storage.

Ex.No :6	Launch and Connect MySQL RDS Database for Inventory Management with CRUD Operations and CloudWatch Monitoring

AIM:

To deploy a **MySQL database instance using Amazon RDS**, connect it to a web application, perform **CRUD operations** (Create, Read, Update, Delete), and monitor the database using **Amazon CloudWatch** metrics.

ALGORITHM:

1. Login to **AWS Management Console** and navigate to **Amazon RDS**.
2. Click **Create Database** and select **MySQL** as the engine.
3. Choose settings: DB instance identifier, credentials, and instance size (e.g., db.t2.micro).
4. Enable public access and configure the **VPC Security Group** to allow inbound MySQL (port 3306).
5. Launch the RDS instance and note the **endpoint** for connection.
6. Use a web app or MySQL client to connect to the database using the endpoint.
7. Create a new database and table for inventory.
8. Perform **CRUD operations** using SQL.
9. Open **Amazon CloudWatch** and view metrics for the RDS instance (e.g., CPUUtilization, FreeStorageSpace).

PROGRAM:

```
-- Create database
CREATE DATABASE InventoryDB;

-- Use database
USE InventoryDB;

-- Create table
CREATE TABLE Products (
   ProductID INT AUTO_INCREMENT PRIMARY KEY,
   ProductName VARCHAR(50),
   Quantity INT,
   Price DECIMAL(10,2)
);

-- Insert records (CREATE)
```

```
INSERT INTO Products (ProductName, Quantity, Price)
VALUES ('Laptop', 10, 50000.00), ('Mouse', 25, 250.00);

-- Retrieve records (READ)
SELECT * FROM Products;

-- Update records (UPDATE)
UPDATE Products SET Quantity = 20 WHERE ProductName = 'Mouse';

-- Delete records (DELETE)
DELETE FROM Products WHERE ProductName = 'Laptop';
```

OUTPUT:

```
mysql> CREATE DATABASE InventoryDB;
Query OK, 1 row affected (0.01 sec)

mysql> USE InventoryDB;
Database changed

mysql> CREATE TABLE Products (
    -> ProductID INT AUTO_INCREMENT PRIMARY KEY,
    -> ProductName VARCHAR(50),
    -> Quantity INT,
    -> Price DECIMAL(10,2)
    -> );
mysql> INSERT INTO Products (ProductName, Quantity, Price)
   -> VALUES ('Laptop', 10, 50000.00), ('Mouse', 25, 250.00);
Query OK, 2 rows affected (0.02 sec)

mysql> SELECT * FROM Products;
+-----------+-------------+----------+----------+
| ProductID | ProductName | Quantity | Price    |
+-----------+-------------+----------+----------+
|         1 | Laptop      |       10 | 50000.00 |
|         2 | Mouse       |       25 |   250.00 |
+-----------+-------------+----------+----------+
2 rows in set (0.00 sec)

mysql> UPDATE Products SET Quantity = 20 WHERE ProductName = 'Mouse';
Query OK, 1 row affected (0.01 sec)
```

RESULT:

The MySQL database was successfully launched using Amazon RDS and connected to a web application. CRUD (Create, Read, Update, Delete) operations were performed on the inventory database, and the changes were reflected as expected. CloudWatch metrics were observed to monitor the performance and resource usage of the RDS instance.

EX.NO:7	Develop a Recovery Management System using Amazon EBS Snapshots and EC2 Instances

AIM:

To back up data from one server and restore it on another using **Amazon EC2** and **Amazon EBS snapshots**, by creating a snapshot, launching a new volume from the snapshot, and attaching it to an EC2 instance for recovery.

ALGORITHM:

1. Launch an **EC2 instance** and create or use an existing **EBS volume** with user data.
2. Create a **snapshot** of the EBS volume to back up the data.
3. Launch a **second EC2 instance** to simulate a recovery server.
4. Create a **new EBS volume** from the snapshot.
5. Attach the new volume to the second EC2 instance.
6. Connect to the instance using **SSH**.
7. Mount the volume to a directory using Linux commands.
8. Access the data and verify recovery was successful.

PROGRAM:

```
# Step 1: Create a snapshot of an existing volume
aws ec2 create-snapshot \
  --volume-id vol-0123456789abcdef0 \
  --description "Backup snapshot of data volume"

# Step 2: Create a new volume from the snapshot
aws ec2 create-volume \
  --availability-zone us-east-1a \
  --snapshot-id snap-0123456789abcdef0 \
  --volume-type gp2

# Step 3: Attach the volume to another EC2 instance
aws ec2 attach-volume \
  --volume-id vol-0abcdef1234567890 \
  --instance-id i-0abcde123456789f0 \
  --device /dev/xvdf
```

Step 4: Connect to EC2 instance and mount volume
ssh -i your-key.pem ec2-user@<instance-public-ip>

# Step 5: On the EC2 instance
sudo mkdir /data
sudo mount /dev/xvdf1 /data

Optional: To auto-mount on reboot, add entry in /etc/fstab

OUTPUT:

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/xvda1       8G  1.5G  6.5G  19% /
/dev/xvdf1      20G  5.0G   15G  25% /data
```

RESULT:

A recovery management system was successfully implemented using **Amazon EBS** and **EC2**. Data was backed up from a volume using a snapshot, and restored to another server by creating a new volume from the snapshot and attaching it to a different EC2 instance. The data was accessed and verified, confirming successful recovery.

EX.NO:8	Add Dynamic Data to a Web Application using Amazon DynamoDB with GSI and Batch Operations

AIM:

To create a DynamoDB table, insert dynamic data using batch and single put operations, retrieve data using **scan** and **query** methods, and use a **Global Secondary Index (GSI)** to perform efficient queries.

ALGORITHM:

1. Open the AWS Management Console and go to **DynamoDB**.
2. Create a **new table** (e.g., `Products`) with a **primary key**.
3. Use the **BatchWriteItem** command to insert multiple records.
4. Use the **Scan** method to retrieve all data from the table.
5. Add a **single item** using the **PutItem** method.
6. Add a **Global Secondary Index (GSI)** to the table on a different attribute.
7. Perform a **query** using the GSI to fetch filtered data.

PROGRAM:

```
# Step 1: Create DynamoDB Table with Global Secondary Index (GSI)
aws dynamodb create-table \
  --table-name Products \
  --attribute-definitions AttributeName=ProductID,AttributeType=S \
                          AttributeName=Category,AttributeType=S \
  --key-schema AttributeName=ProductID,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --global-secondary-indexes '[
    {
      "IndexName": "CategoryIndex",
      "KeySchema": [
        {"AttributeName": "Category", "KeyType": "HASH"}
      ],
      "Projection": {"ProjectionType": "ALL"},
      "ProvisionedThroughput": {
      "ReadCapacityUnits": 5,
        "WriteCapacityUnits": 5
      }
```

```
  ]'

# Step 2: Batch Insert Multiple Records
aws dynamodb batch-write-item --request-items '{
  "Products": [
    {
      "PutRequest": {
        "Item": {
          "ProductID": {"S": "101"},
          "ProductName": {"S": "Keyboard"},
          "Category": {"S": "Electronics"}
        }
      }
    },
    {
      "PutRequest": {
        "Item": {
          "ProductID": {"S": "102"},
          "ProductName": {"S": "Shoes"},
          "Category": {"S": "Apparel"}
        }
      }
    }
  ]
}'

# Step 3: Add a Single Record using PutItem
aws dynamodb put-item \
  --table-name Products \
  --item '{
    "ProductID": {"S": "103"},
    "ProductName": {"S": "Mouse"},
    "Category": {"S": "Electronics"}
  }'

# Step 4: Retrieve All Records using Scan
aws dynamodb scan --table-name Products

# Step 5: Query Records using the Global Secondary Index (GSI)
aws dynamodb query \
  --table-name Products \
  --index-name CategoryIndex \
  --key-condition-expression "Category = :cat" \
  --expression-attribute-values '{":cat":{"S":"Electronics"}}'
```

OUTPUT:


```
{
    "Items": [
        {
            "ProductID": {"S": "101"},
            "ProductName": {"S": "Keyboard"},
            "Category": {"S": "Electronics"}
        },
        {
            "ProductID": {"S": "103"},
            "ProductName": {"S": "Mouse"},
            "Category": {"S": "Electronics"}
        }
    ],
    "Count": 2
}
```


RESULT

A DynamoDB table named **Products** was successfully created. Records were inserted using both **batch** and **single** insert methods. Data was retrieved using **scan** and **query** operations. A **Global Secondary Index (GSI)** was created on the **Category** attribute, and it was used to query filtered data efficiently from the table.

EX.NO:9	Create and Optimize an AWS Lambda Function with S3 Trigger to Monitor Image Uploads

AIM:

To create an AWS **Lambda function** that is triggered by an **Amazon S3** image upload event, configure the function's memory settings for optimal performance, and monitor logs and metrics using **Amazon CloudWatch**.

ALGORITHM:

1. Create an **S3 bucket** to store uploaded images.
2. Create a **Lambda function** in Python to log file uploads.
3. Grant the Lambda function **permissions** to read from S3 and write logs to CloudWatch.
4. Configure the **S3 trigger** to invoke the Lambda function on object creation.
5. Upload an image to the S3 bucket to trigger the function.
6. Optimize the Lambda function by adjusting **memory size and timeout** settings.
7. Monitor **CloudWatch logs** and **Lambda metrics** for performance analysis.

PROGRAM:

```
# Step 1: Create an S3 bucket
aws s3api create-bucket --bucket image-upload-lambda-demo --region us-east-1

# Step 2: Create IAM Trust Policy for Lambda
cat > trust-policy.json <<EOF
{
 "Version": "2012-10-17",
 "Statement": [{
  "Effect": "Allow",
  "Principal": {"Service": "lambda.amazonaws.com"},
  "Action": "sts:AssumeRole"
 }]
}
EOF

# Step 3: Create IAM Role for Lambda execution
aws iam create-role \
 --role-name LambdaS3ExecutionRole \
 --assume-role-policy-document  file://trust-policy.json
```

```
aws iam attach-role-policy \
  --role-name LambdaS3ExecutionRole \
  --policy-arn arn:aws:iam::aws:policy/AWSLambdaExecute

# Step 5: Create Lambda function code
cat > lambda_function.py <<EOF
import json

def lambda_handler(event, context):
    print("Event Received:", json.dumps(event))
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = event['Records'][0]['s3']['object']['key']
    print(f"New object uploaded: {key} in bucket: {bucket}")
    return {
        'statusCode': 200,
        'body': json.dumps('Success')
    }
EOF

# Step 6: Zip the Lambda function
zip function.zip lambda_function.py

# Step 7: Create Lambda function (replace <account-id> as needed)
aws lambda create-function \
  --function-name S3ImageLogger \
  --runtime python3.9 \
  --role arn:aws:iam::<your-account-id>:role/LambdaS3ExecutionRole \
  --handler lambda_function.lambda_handler \
  --zip-file fileb://function.zip \
  --timeout 15 \
  --memory-size 256

# Step 8: Allow S3 to invoke Lambda
aws lambda add-permission \
  --function-name S3ImageLogger \
  --statement-id s3invoke \
  --action "lambda:InvokeFunction" \
  --principal s3.amazonaws.com \
  --source-arn  arn:aws:s3:::image-upload-lambda-demo

# Step 9: Add S3 trigger to Lambda
aws s3api put-bucket-notification-configuration \
  --bucket image-upload-lambda-demo \
  --notification-configuration '{
    "LambdaFunctionConfigurations": [{
      "LambdaFunctionArn": "arn:aws:lambda:us-east-1:<your-account-
id>:function:S3ImageLogger",
```

```
    "Events": ["s3:ObjectCreated:*"]
  }]
}'

# Step 10: Upload an image to trigger Lambda
aws s3 cp sample.jpg s3://image-upload-lambda-demo/
```

OUTPUT:

START RequestId: 84c6a847-b0d7-4b3d-b173-328c4f73d8d7 Version: $LATEST

Event Received: {"Records":[{"s3":{"bucket":{"name":"image-upload-lambda-demo"},"object":{"key":"sample.jpg"}}}]}

New object uploaded: sample.jpg in bucket: image-upload-lambda-demo

END RequestId: 84c6a847-b0d7-4b3d-b173-328c4f73d8d7

REPORT RequestId: 84c6a847-b0d7-4b3d-b173-328c4f73d8d7

Duration: 127.54 ms

Billed Duration: 200 ms

Memory Size: 256 MB

Max Memory Used: 79 MB

Init Duration: 143.21 ms

RESULT:

A Lambda function was successfully created and linked with an S3 bucket trigger. Upon uploading an image to the S3 bucket, the Lambda function was invoked automatically. The function logged the upload event, and performance metrics such as duration and memory usage were observed in **CloudWatch**, confirming successful setup and optimization.

EX.NO:10	Create an AWS Glue ETL Job to Transform S3 CSV Data and Load into Amazon Redshift with Scheduled Execution

AIM:

To build an **ETL pipeline using AWS Glue**, extract data from a **CSV file stored in S3**, transform it, and load the output into an **Amazon Redshift** table. Additionally, schedule the job to run daily or weekly using AWS Glue job scheduling.

ALGORITHM:

1. Upload a CSV file to an **Amazon S3 bucket**.
2. Create an **AWS Glue Crawler** to detect the schema of the source CSV data.
3. Set up an **AWS Glue connection** to the **Amazon Redshift cluster**.
4. Create a **target table** in Amazon Redshift to receive the transformed data.
5. Create a **Glue ETL job** using AWS Glue Studio or script editor.
6. Use a Python/Scala script to transform the data as needed.
7. Choose Redshift as the **target**, and provide necessary Redshift connection and table details.
8. Run the ETL job and verify the data is loaded into Redshift.
9. Create a **Glue job trigger** to schedule the job to run **daily/weekly**.
10. Verify schedule using Glue console or CloudWatch Events.

PROGRAM:

```
import sys
from awsglue.transforms import *
from awsglue.utils import getResolvedOptions
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.job import Job

# Glue boilerplate
args = getResolvedOptions(sys.argv, ['JOB_NAME'])
sc = SparkContext()
glueContext = GlueContext(sc)
spark = glueContext.spark_session
job = Job(glueContext)
job.init(args['JOB_NAME'], args)
```

```
# Load data from S3
datasource = glueContext.create_dynamic_frame.from_options(
    connection_type="s3",
    connection_options={"paths": ["s3://your-source-bucket/data.csv"]},
    format="csv",
    format_options={"withHeader":  True}
)

# Optional: Apply transformations
transformed_data = ApplyMapping.apply(
    frame=datasource,
    mappings=[
        ("id", "string", "id", "int"),
        ("name", "string", "name", "string"),
        ("price", "string", "price", "double")
    ]
)

# Write data to Redshift
glueContext.write_dynamic_frame.from_jdbc_conf(
    frame=transformed_data,
    catalog_connection="redshift-conn",
    connection_options={
        "dbtable": "public.transformed_products",
        "database": "redshiftdb"
    },
    redshift_tmp_dir="s3://your-temp-dir/"
)

job.commit()
```

OUTPUT:

```
Job Name: s3-to-redshift-etl
Status: SUCCEEDED
Start Time: 2025-07-13 08:00:12 UTC
End Time: 2025-07-13 08:02:31 UTC
Records Read: 1000
Records Written to Redshift: 1000
Target Table: public.transformed_products
Schedule: Daily at 08:00 UTC
```

RESULT:

An AWS Glue ETL job was successfully created to transform CSV data from an S3 bucket and load it into an Amazon Redshift cluster. The job was tested and verified to load the data correctly. A scheduled trigger was configured to run the ETL job **daily**, and scheduling was confirmed in the Glue console.

EX.NO:11	Detect and Analyze Text in Images using Amazon Rekognition and Amazon S3

AIM:

To create an Amazon S3 bucket, upload image files, and use **Amazon Rekognition** to detect and analyze **text content** from the uploaded images.

ALGORITHM:

1. Create a new **S3 bucket** to store images.
2. Upload image files (containing text) to the S3 bucket.
3. Use **Amazon Rekognition's DetectText API** to analyze the images.
4. Retrieve and display the text detected from the image.
5. Interpret the output to understand the text content.

PROGRAM:

```
# Step 1: Create a new S3 bucket
aws s3api create-bucket --bucket image-text-analysis-2025 --region us-east-1

# Step 2: Upload an image with text to the bucket
aws s3 cp sample-text-image.jpg s3://image-text-analysis-2025/

# Step 3: Detect text in the uploaded image using Amazon Rekognition
aws rekognition detect-text \
  --image  "S3Object={Bucket=image-text-analysis-2025,Name=sample-text-image.jpg}"  \
  --region us-east-1
```

OUTPUT:


```
{
  "TextDetections": [
    {
      "DetectedText": "WELCOME TO AI LAB",
      "Type": "LINE",
      "Confidence": 98.7
    },
    {
      "DetectedText": "AI",
      "Type": "WORD",
      "Confidence": 99.3
    }
  ]
}
```


RESULT:

The image was successfully uploaded to the Amazon S3 bucket, and **Amazon Rekognition** detected and extracted text from the image. The detected text was listed with high confidence scores, confirming the successful use of AWS Rekognition for image text analysis.

EX.NO:12	Data Preprocessing using Amazon SageMaker and Amazon S3

AIM:

To create an S3 bucket for storing a dataset, launch a **SageMaker Notebook Instance**, load the dataset, and perform **data preprocessing** steps including handling missing values, encoding categorical variables, and feature scaling using **Pandas**, **NumPy**, and **Scikit-learn**.

ALGORITHM:

1. Create an **Amazon S3 bucket** and upload the dataset (CSV file).
2. Launch a **SageMaker notebook instance**.
3. Select a **VPC** and **subnet** during notebook creation for networking.
4. Open Jupyter Notebook in the instance.
5. Import required Python libraries (pandas, numpy, sklearn).
6. Load dataset from S3 using the **Boto3** or **S3FS** connector.
7. Perform preprocessing:
8. Handle **missing values**.
9. Encode **categorical features**.
10. **Normalize** or **scale** numerical features.

PROGRAM:

```
# Step 1: Import Libraries
import boto3
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder, StandardScaler
from io import StringIO
# Step 2: Connect to S3
s3 = boto3.client('s3')
bucket = 'my-dataset-bucket-2025'
key = 'data.csv'
# Step 3: Load CSV from S3 into DataFrame
response = s3.get_object(Bucket=bucket, Key=key)

df = pd.read_csv(response['Body']
```

```
# Step 5: Encode categorical variables
label_encoders = {}
for col in df.select_dtypes(include=['object']).columns:
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col].astype(str))
    label_encoders[col] = le

# Step 6: Scale/Normalize features
scaler = StandardScaler()
scaled_features = scaler.fit_transform(df)
df_scaled = pd.DataFrame(scaled_features, columns=df.columns)

# Step 7: Display preprocessed data
df_scaled.head()
```

OUTPUT:

	feature1	feature2	category_encoded	target
0	-0.234	1.278	0.0	1.0
1	0.458	-0.923	1.0	0.0
2	1.208	0.164	2.0	1.0

RESULT:

The dataset was successfully loaded from Amazon S3 into a SageMaker notebook instance. Necessary preprocessing steps such as missing value handling, categorical encoding, and feature scaling were performed using Pandas, NumPy, and Scikit-learn. The processed dataset is ready for model training or analysis.