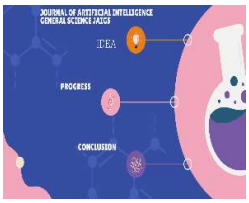
Vol. 02, Issue 01, February 2024
Journal of Artificial Intelligence General Science JAIGS

https://ojs.boulibrary.com/index.php/JAIGS



Automate Amazon Aurora Global Database Using Cloud Formation

Mr. Kondala Rao Patibandla

Senior Software Engineer, Aircraft Technical & Operations, Southwest Airlines,

Irving, TX 75038

ARTICLEINFO

Article History:

Received: 10.02.2024

Accepted: 15.02.2024

Online: 29.02.2024

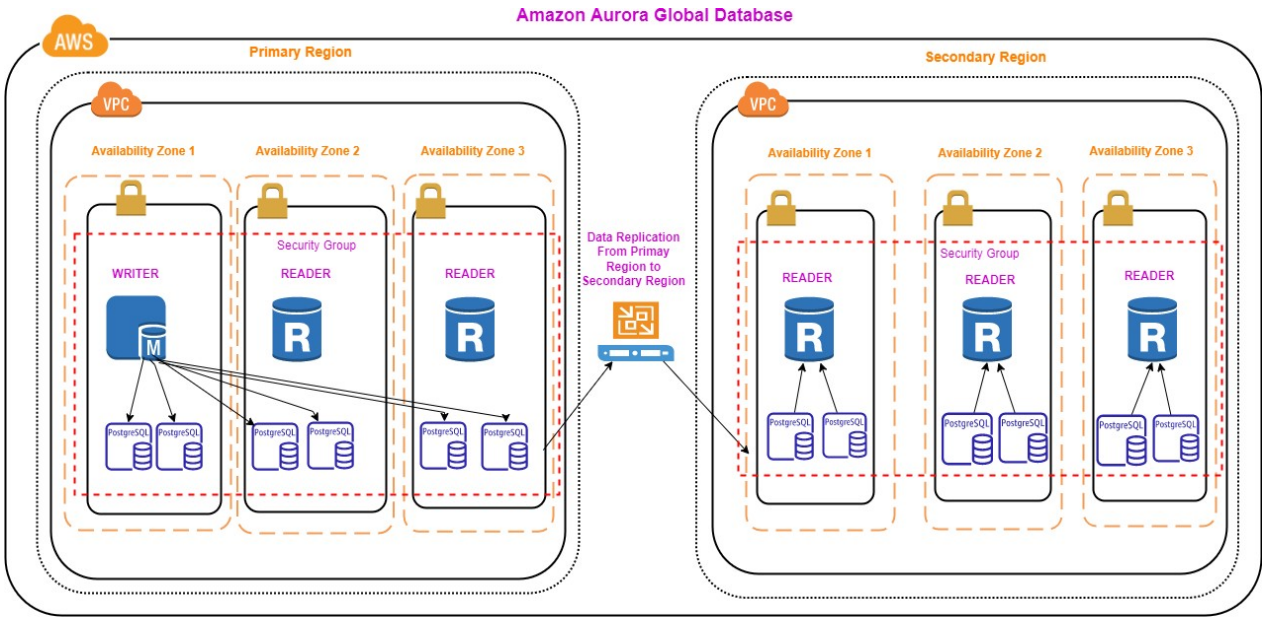
Keyword: AWS RDS, Aurora RDS, Global Database, RDS cluster, Aurora Global Database

ABSTRACT

This article provides an in-depth guide on deploying an AWS Aurora Global Database using AWS CloudFormation. It covers the benefits of Aurora Global Databases, such as high availability, low-latency global reads, and disaster recovery capabilities. The article details the CloudFormation template structure and key parameters needed to set up the Aurora Global Database, enabling seamless data replication across multiple AWS regions. Practical examples and best practices are included to ensure a robust and efficient deployment process.

I.Introduction

Amazon Aurora Global Database is designed for globally distributed cloud applications in AWS. It provides high availability and database resiliency by way of its ability to fail over to another AWS region. It allows a database to span multiple regions (AWSlimits regions to a maximum of six), and it consists of one primary and up to five secondary regions in a global database cluster.Primary region can perform read and write operations, whereas the second region can perform read operations only. The way AWS facilitates this feature is by activating writer endpoints in the primary region and deactivating writer endpoints in secondary regions. Furthermore,Aurora replicates data from primary region to secondary regions, usually under a second.



Ahigh-levelillustration ofAuroraGlobal Database.

II. Prerequisites

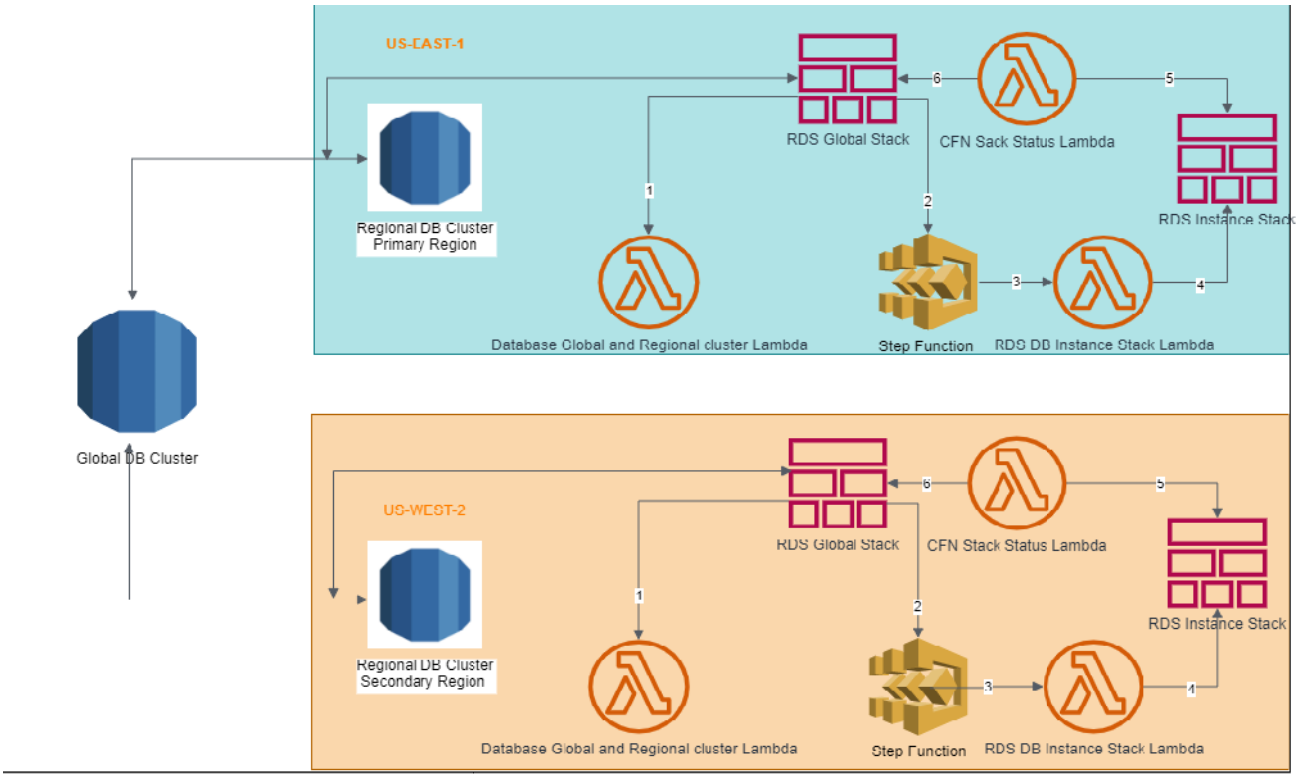
To deploy this solution, you must have the following prerequisites:

- An AWS account.
- AWS CLI with administrator permissions.
- Python 3, preferably the latest version.
- Basic knowledge ofAWS SDK forpython (boto3).
- Basic knowledge of CloudFormation templates.
- Basic knowledge of Lambda and Step functions.

III. Creating an RDS Global Database

In order to create an RDS global database, we need to define global and regional database clusters. Wethenneed todefinedatabaseinstances in each regional cluster.

Let us keep in mind that in order to define an RDS global database, we need to Subnet Group, RDS Security group & DBParameters group.



AsampleAmazonAuroraGlobalDatabasetopology.

The sample representation of anAmazonAurora Global Database topology depicted above involves the following componentsand resources in its setup:

1. RDS Global Stack

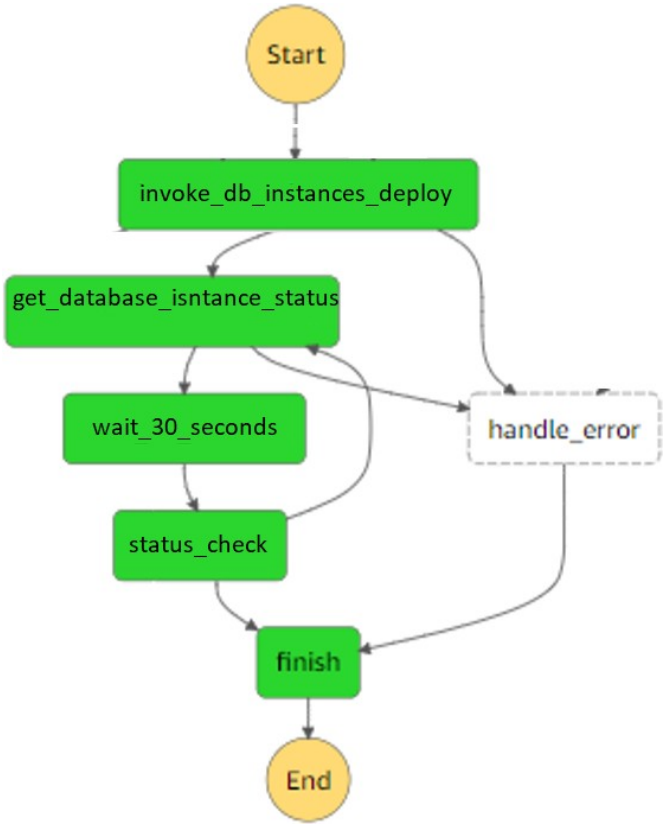
This is the base CloudFormation (CFN) stack to create RDSAurora Global, regional database clusters,and instances in each regional cluster.This stack defines RDS subnet, Database Global and Regional cluster Lambda, StepFunction, RDS DB instances stack Lambda & CFN stack status Lambda as resources to be created.

2. Database Global and Regional ClusterLambda

This Lambda creates regional database clusters first, and it then creates a globaldatabaseclusterbyassigningthenewly createdregionalclusterstotheglobal cluster.

3. Step Function

This state machine is responsible for creating database instances stack as a task, waiting and checking thestatus of this task until completion.



4. RDS DB Instance Stack Lambda

This Lambda is responsible for creating a CloudFormation stack that creates databaseinstances.

5. CFN Stack Status Lambda

This Lambda is responsible for checking the RDS instances stack's status and returning thestatus to the Step Function.

All of the above resources are defined in the 'global-rds.yaml' CFN template. Code snippets for these resources are given below. For ease of reference, the individual code snippets carry the same number as the resources explained above.

AWSCLIcommandstodeploycloudformationtemplate:

```
# Deploy database cluster in primary region
aws cloudformation create-stack --region=us-east-1
--stack-name global-db-east-1 --template-body global-rds.yaml
--parameters pPrivateSubnetId1=<your private subnet1>
pPrivateSubnetId2=<your private subnet2> pPrivateSubnetId3=<your private subnet3>
pDatabaseInstanceClass=db.r5.large pDatabaseEngineType=aurora-postgresql
pDatabaseEngineVersion=14.x

# Deploy database cluster in secondary region
aws cloudformation create-stack --region=us-west-2
--stack-name global-db-east-1 --template-body global-rds.yaml
--parameters pPrivateSubnetId1=<your private subnet1>
pPrivateSubnetId2=<your private subnet2> pPrivateSubnetId3=<your private subnet3>
pDatabaseInstanceClass=db.r5.large pDatabaseEngineType=aurora-postgresql
pDatabaseEngineVersion=14.x
```

1. RDS Global Stack

YAML

```yaml
AWSTemplateFormationVersion: "2010-09-09"
Transform: "AWS::Serverless-2016-10-31"
Description: AWS

Parameters:
  pPrivateSubnetId1:
    Description: AWS RDS Global DB subnet 1 Goupd Id
    Type: String

  pPrivateSubnetId2:
    Description: AWS RDS Global DB subnet 2 Goupd Id
    Type: String

  pPrivateSubnetId3:
    Description: AWS RDS Global DB subnet 3 Goupd Id
    Type: String

 pDatabaseInstanceClass:
    Description: Database Instance Type
    Type: String

 pDatabaseEngineType:
    Description: Database Engine Type
    Type: String

 pDatabaseEngineVersion:
    Description: Database Engine Version
    Type: String

 Resources:

```

2. Database Global and Regional Cluster Lambda

```python
import os
import boto3

def handler(event, context):
  resource_properties = even.get("ResourceProperties")
  # Create database regional cluster first
  cluster_arn = create_db_regional_cluser(resource_properties)
  # Create database global cluster with regional cluster id
  create_global_cluster(resource_properties, cluster_arn)
  return True

def get_rds_client(region):
  return boto3.client('rds', region)

def create_global_cluster(resource_properties, cluster_arn):
  rds_client = get_rds_client(resource_properties.get('Region'))
  rds_client.create_global_cluster(
    GlobalClusterIdentifier=resource_properties.get('GloablClusterId'),
    SourceDBClusterIdentifier=cluster_arn
  )

def create_db_regional_cluster(resource_properties):
  rds_client = get_rds_client(resource_properties.get('Region'))
  response = rds_client.create_db_cluster(
    DBClusterIdentifier=resource_properties.get('ClusterId'),
    Engine=resource_properties.get('Engine'),
    EngineVersion=resource_properties.get('EngineVersion'),
    Port=resource_properties.get('Port')
  )
  return response.get('DBCluster').get('DBClusterArn')

```

3. RDS DB Instance Stack Lambda

```python
import boto3

def handler(event, context):
  stack_name = event.get('StackName')
  region = event.get('Region')
  params = event.get('Parameters')
  params['pDatabaseParameterGroup'] = get_rds_params_group(region)
  params['pDatabseSubnetGroup'] = get_rds_subnet_group(region)

def get_cfn_client(region):
  return boto3.client('cloudformation', region)

def get_rds_client(region):
  return boto3.client('rds', region)

def create_databse_instances(stack_name, params, region, template_path):
  get_cfn_client(region).create_stack(
    StackName=stack_name,
    TemplateBody=parse_template(template_path)
    Parameters=Params,
    Capabilities=['CAPABILITY_AUTO_EXPAND']
  )

def parse_template(template_path, region):
    with open(template) as template_file:
        data = template_file.read()
    get_cfn_client(region).validate_template(Template=data)
    return data

def get_rds_params_group(region):
  paras_group = []
  paginator = get_rds_client(region).get_paginator('descrip_db_cluster_parameter_group')
  for grouppage in paginator.paginator()
    paras_group =return_list+ grouppage.get('DBClusterParameterGroup')
  return paras_group

def get_rds_subnet_group(region):
  subnet_group = []
  paginator = get_rds_client(region).get_paginator('describe_db_subnet_group')
  for grouppage in paginator.paginator()
    subnet_group =return_list+ grouppage.get('DBSubnetGroup')
   return subnet_group
```

4. RDS Instance Stack

```yaml
AWSTemplateFormationVersion: "2010-09-09"
Transform: "AWS::Serverless-2016-10-31"
Description: AWS

Parameters:
  pDatabaseInstanceClass:
    Description: Database Instance Class
    Type: String

  pDatabaseSubentGroup:
    Description: Database Subnet Group
    Type: String

  pDatabaseParameterGroup:
    Description: Database Parameter Group
    Type: String

Resources:
 rPrimaryDatabaseInstance:
    Type: AWS::RDS::DBInstance
    Properties:
        DBInstanceIdentifier: !Sub 'db-instance-${AWS::Region}-1'
        DBClusterIdentifier: !Sub regional-db-cluster-${AWS::Region}
        DBInstanceClass: !Ref pDatabaseInstanceClass
        DBSubnetGroupName: !Ref pDatabaseSubentGroup
        DBParameterGroup: !Ref pDatabaseParameterGroup
        Engine: aurora-Postgresql

 rReplicationDatabaseInstance1:
    Type: AWS::RDS::DBInstance
    Properties:
        DBInstanceIdentifier: 'db-instance-${AWS::Region}-2'
        DBClusterIdentifier: !Sub test-cluster-${AWS::Region}
        DBInstanceClass: !Ref pDatabaseInstanceClass
        DBSubnetGroupName: !Ref pDatabaseSubentGroup
        DBParameterGroup: !Ref pDatabaseParameterGroup
        Engine: aurora-Postgresql

 rReplicationDatabaseInstance2:
    Type: AWS::RDS::DBInstance
    Properties:
        DBInstanceIdentifier: 'db-instance-${AWS::Region}-3'
        DBClusterIdentifier: !Sub test-cluster-${AWS::Region}
        DBInstanceClass: !Ref pDatabaseInstanceClass
        DBSubnetGroupName: !Ref pDatabaseSubentGroup
        DBParameterGroup: !Ref pDatabaseParameterGroup
```

5. CFN Stack Status Lambda

```python
import boto3

def handler(event, context):
  stack_name = event.get('StackName')
  region = event.get(''Region)
  stack_status = get_stack_status(stack_name, region)
  if statck_status == 'CREATE_IN_PROGRESS'
    return 'WAIT'
  if stack_staus == 'CREATE_COMPLETE'
    return 'SUCCESS'

def get_cfn_client(region):
  return boto3.client('cloudformation', region)

def get_stack_status(stack_name, region):
  stack_response = get_cfn_client(region).describe_stacks(
    StackName=stack_name
  ).get('Stack')

  if stack_response:
    stack_status = stack_response[0].get('StackStatus')

  return stack_status
```

When all the steps defined above are completed successfully, one can see the newly created Amazon Aurora Global PostgreSQL Database, as shown below.

DB identifier	DB cluster identifier	Role	Engine	Region & AZ	Size	Status
global-db-cluster	global-db-cluster	Global database	Aurora PostgreSQL	2 regions	2 clusters	Available
regional-db-cluster-us-east-1	regional-db-cluster-us-east-1	Primary cluster	Aurora PostgreSQL	us-east-1	3 instances	Available
db-instance-us-east-1-1	regional-db-cluster-us-east-1	Writer instance	Aurora PostgreSQL	us-east-1b	db.r5.large	Available
db-instance-us-east-1-2	regional-db-cluster-us-east-1	Reader instance	Aurora PostgreSQL	us-east-1c	db.r5.large	Available
db-instance-us-east-1-3	regional-db-cluster-us-east-1	Reader instance	Aurora PostgreSQL	us-east-1a	db.r5.large	Available
regional-db-cluster-us-west-2	regional-db-cluster-us-west-2	Secondary cluster	Aurora PostgreSQL	us-west-2	3 instances	Available
db-instance-us-west-2-1	regional-db-cluster-us-west-2	Reader instance	Aurora PostgreSQL	us-west-2c	db.r5.large	Available
db-instance-us-west-2-2	regional-db-cluster-us-west-2	Reader instance	Aurora PostgreSQL	us-west-2b	db.r5.large	Available
db-instance-us-west-2-3	regional-db-cluster-us-west-2	Reader instance	Aurora PostgreSQL	us-west-2a	db.r5.large	Available

IV. Fail-Over Scenario

With Aurora Global Database, one can expect two failover scenarios – managed planned failover and unplanned failover.

Managed Planned Fail-Over

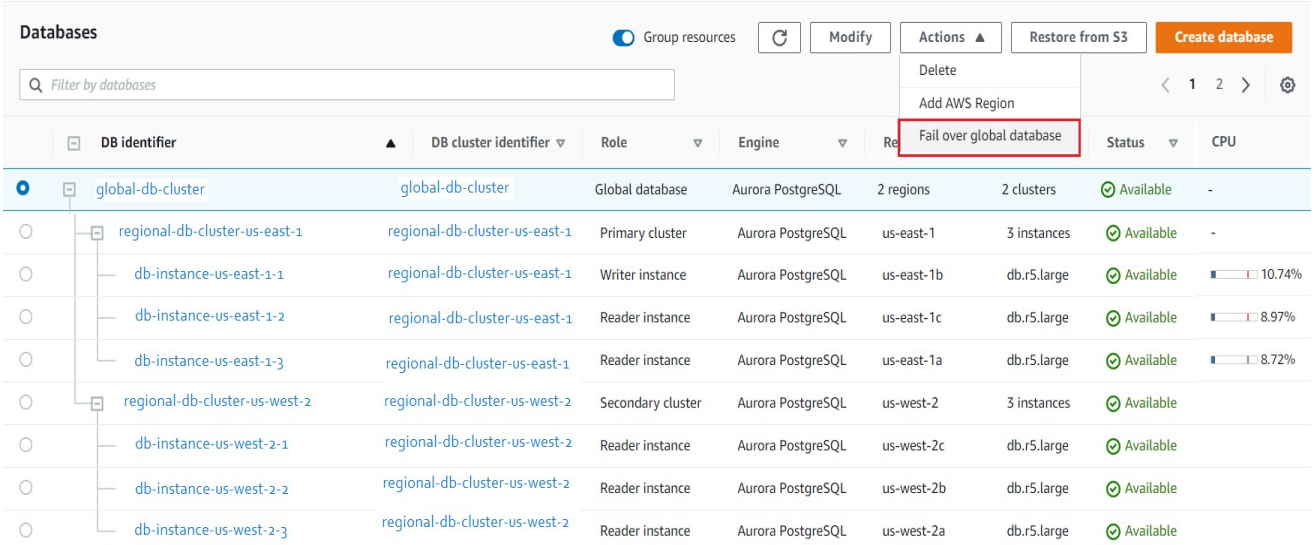
A managed planned fail-over scenario works best when both the regions of the global cluster are in normal operation. When performing this operation, the writer endpoint in the active region is replaced with a reader endpoint. Vice-versa happens in the passive region, i.e., the reader endpoint in the passive region is replaced with the writer endpoint. This ensures that active and passive regions are flipped after performing the fail-over operation.

Planned fail-over can be performed in multiple ways. Some of the ways are:

- Using AWS console
- AWS CLI
- Scripts that use AWS SDK
- AWS CDK

Using AWS Console

The picture below depicts options to select in the AWS console's 'Databases' section on the 'RDS' page.



AW CLI

Execute the command given below to perform managed planned fail-over using AWS CLI.

```
awsrds --region us-east-1 failover-global-cluster
--global-cluster-identifier global-db-cluster
--target-db-cluster-identifier arn:aws:rds:us-west-2:{AWS Account Number}:cluster:db-regional-cluster-us-west-2
```

Unplanned Fail-Over

We perform unplanned fail-over when the current active database cluster goes down. The following steps need to be performed:

- Remove the passive region (secondary region) database cluster from the global cluster. After removing it from the global database cluster, this works as a stand-alone database cluster, and one of the reader instances turns into a writer instance. We can assign it back to the global cluster, allowing us to perform write and read operations on a stand-alone database cluster.
- Delete the affected database cluster, which was running as an active cluster in the global database once the affected AWS region is operational. Then, assign a stand-alone cluster to the global database as an active region cluster. Finally, create a new secondary database cluster in the previously affected region and assign it to the global database cluster as a passive region cluster.

V. Conclusion

I have defined comprehensive steps which would create and configure an Amazon Aurora Global Database setup. This would provide a database with high availability and fault tolerance. This database setup can cater to a multi-regional application setup, making it resilient to failures. We also provided steps to automate and simplify creating a complex global database setup.