Documentation

AWS Additional Service Integration using AWS CDK

Manjitha Manchanayake

AWS CDK

AWS CDK allows you to define and provision AWS infrastructure using familiar programming languages such as Python, TypeScript, Java, C#, and more. Here I'm using Python.

AWS CDK Installation

Prerequisites:

- AWS CLI
- Node.js
- Python
- IDE for programming language (VS Code)

After installing the prerequisites cdk toolkit can be installed
npm install -g aws-cdk

Check the cdk has this version
$ cdk –version
2.120.0

If you use Windows, be sure Python is on your PATH. To see if it is, type python at a command prompt. The easiest way to make sure Python is on your PATH is to tick the Add Python 3.x to PATH checkbox on the first screen of the Python installer wizard. If you already have Python installed, but it's not on your PATH, you can add it by editing the PATH environment variable. To edit environment variables, click the Environment Variables button in the Advanced page of Windows' System Properties. (Quickest way to get there: press and release the Windows key, type env, and choose Edit the system environment variables from the Start menu.)

1. Create an empty directory on your system:
   mkdir cdk_workshop && cd cdk_workshop

2. We will use cdk init to create a new Python CDK project:
   cdk init sample-app --language python

3. Create Virtual environment.
   $ python3 -m venv .venv

4. After the init process completes and the virtualenv is created, activate the virtualenv.
   $ source .venv/bin/activate

   If Windows platform:
   % .venv\Scripts\activate.bat

5. Once the virtualenv is activated, you can install the required dependencies.
   $ pip install -r requirements.txt

Project Directory

.venv - The python virtual environment information

cdk_workshop — A Python module directory.

cdk_workshop.egg-info - Folder that contains build information relevant for the packaging on the project

cdk_workshop_stack.py—A custom CDK stack construct for use in your CDK application.

tests — Contains all tests.

unit — Contains unit tests.

test_cdk_workshop.py—A trivial test of the custom CDK stack created in the cdk_workshop package. This is mainly to demonstrate how tests can be hooked up to the project**.**

app.py — The "main" for this sample application.

cdk.json — A configuration file for CDK that defines what executable CDK should run to generate the CDK construct tree.

README.md — The introductory README for this project.

requirements.txt—This file is used by pip to install all of the dependencies for your application.

Why we need a stack?

To logically group different sections of deployment so we can add different resources.

A stack is a unit of deployment, and it represents a single CloudFormation stack when deployed. This means that all the resources defined within a stack are deployed together and can be managed as a single unit.

My stack “combination_rds_stack” contains below resources.

- VPC
- AWS RDS
- SNS
- SQS
- Cloudwatch
- AWS Lambda

CDK Constructs and Modules

Here the required AWS CDK constructs and modules are imported.

```
# Import necessary constructs from AWS CDK and AWS services
from aws_cdk import (
    CfnOutput,
    aws_secretsmanager as secretsmanager,
    Duration,
    aws_eks as aws_eks,
    Stack,
    aws_ec2 as ec2,
    aws_iam as aws_iam,
    aws_rds as rds,
    aws_sqs as sqs,
    aws_sns as sns,
    aws_sns_subscriptions as subs,
    aws_s3 as s3,
    aws_lambda as lambda_,
    RemovalPolicy
)
from constructs import Construct
from aws_cdk.aws_sns import (
    Topic
)
from aws_cdk.aws_sqs import (
    Queue
)
from aws_cdk.aws_sns_subscriptions import (
    SqsSubscription,
    EmailSubscription
)
from aws_cdk.aws_cloudwatch import (
    Alarm,
    Metric
)
from aws_cdk.aws_cloudwatch_actions import (
    AutoScalingAction,
    SnsAction
)
from aws_cdk.aws_sns import (
```

```
    Topic
)
from aws_cdk.aws_sqs import (
    Queue
)
from aws_cdk.aws_sns_subscriptions import (
    SqsSubscription,
    EmailSubscription
)
from aws_cdk.aws_logs import (
    LogGroup,
    LogStream
)
```

Then Class “AwsCaaSProjectStack” is defined.

VPC

- For the testing I created custom vpc in “combinationRDSstack” with CIDR 10.1.0.0/16 to store RDS instance

4 Subnets

Availability Zone 1 - eu-central-1a
public subnet2 - 10.1.0.0/24
private subnet2 - 10.1.2.0/24

Availability Zone 2 – eu-central-1b
public subnet1 - 10.1.1.0/24
private subnet1 - 10.1.3.0/24

```
#  Create a custome VPC
        custom_vpc = ec2.Vpc(
            self, "customvpc",
            ip_addresses=
ec2.IpAddresses.cidr(Prod_configs['vpc_config']['vpc_cidr']),
            max_azs= 2,
            subnet_configuration=[
                ec2.SubnetConfiguration(
                    name="PublicSubnet",
cidr_mask=Prod_configs["vpc_config"]["cidr_mask"],
subnet_type=ec2.SubnetType.PUBLIC
                ),
                ec2.SubnetConfiguration(
                    name="PrivateSubnet",
cidr_mask=Prod_configs["vpc_config"]["cidr_mask"],
subnet_type=ec2.SubnetType.PRIVATE_ISOLATED
                ),
            ])
```

- After the testing, I used the VPC of the EKS cluster. (10.0.0.0/16)

2 availability zones and 3 subnets per one availability zone. Totally 6 subnets were created.

eu-central-1a
Public Subnet - 10.0.0.0/24
Private subnet 1 - 10.0.2.0/24
Private subnet 2 for RDS - 10.0.4.0/24

eu-central-1b
Public Subnet - 10.0.1.0/24
Private subnet 1 - 10.0.3.0/24
Private subnet 2 for RDS - 10.0.5.0/24

PRIVATE_WITH_EGRESS subnets allow outbound internet connectivity for instances, typically through a NAT gateway or NAT instance.

PRIVATE_ISOLATED subnets do not allow outbound internet connectivity, providing a higher level of isolation for resources that should not have direct access to the internet.

```
availability_zones = ['eu-central-1a', 'eu-central-1b'] # Define the availability zones

        # Creation of custom VPC
        vpc = ec2.Vpc(self, 'Vpc',
            ip_addresses=ec2.IpAddresses.cidr("10.0.0.0/16"),
            vpc_name='eks-vpc',
            enable_dns_hostnames=True,
            enable_dns_support=True,
            availability_zones=availability_zones,
            subnet_configuration=[
                 # Subnet for public resources accessible from the internet
                ec2.SubnetConfiguration(
                    name='eks-subnet-public',
                    subnet_type=ec2.SubnetType.PUBLIC,
                    cidr_mask=24
                ),
                # Subnet for private resources with internet access for outbound traffic
                ec2.SubnetConfiguration(
                    name='eks-subnet-private',
                    subnet_type=ec2.SubnetType.PRIVATE_WITH_EGRESS,
                    cidr_mask=24
                ),
                # Subnet for private resources isolated from the internet
                ec2.SubnetConfiguration(
                    name='eks-subnet-privateRDS',
                    subnet_type=ec2.SubnetType.PRIVATE_ISOLATED,
                    cidr_mask=24
                )
            ]
        )
```

RDS

Why we need RDS? Applications running on EKS may require a relational database to store and manage data. So, when I give my RDS endpoint, to the application, it will automatically store data. the WordPress application is configured to use an Amazon RDS instance as its database.

DB in EC2?

- ✓ Access to Db in EC2.
- ✓ If we need to have specific OS version, we need to install DB in EC2

Why shouldn't we run DBs on EC2?

- ✓ Lot of things to manage (admin overhead)
- ✓ Have concerns on backup and DR
- ✓ EC2 is running in a single AZ
- ✓ Have to setup monitoring manually.

- Security group for RDS

```
# Create a security group for the RDS instance
    rds_security_group = ec2.SecurityGroup(
        self, 'RDSSecurityGroup',
        vpc=vpc,  # Use the VPC created earlier
        allow_all_outbound=True  # Adjust outbound rules based on your requirements
    )

    # Allow inbound connections on the MySQL port (3306)
    rds_security_group.add_ingress_rule(
        peer=ec2.Peer.ipv4('0.0.0.0/0'),
        connection=ec2.Port.tcp(3306),
        description='Allow inbound MySQL connections'
    )
```

- ✓ Security group named 'RDSSecurityGroup' is created for the RDS instance.
- ✓ Outbound rule - allows all outbound traffic from the RDS instance.
- ✓ inbound rule - allowing traffic from any IPv4 address (0.0.0.0/0) on the MySQL port (3306).

- Create RDS Database instance

```
        #RDS Instance Deployment
        myDB = rds.DatabaseInstance(self,
                                    "MyDatabase",
                                    engine= rds.DatabaseInstanceEngine.MYSQL,
                                    vpc= vpc,
                                    vpc_subnets= ec2.SubnetSelection(
                                        #subnet_type= ec2.SubnetType.PUBLIC,
                                        subnet_type=
ec2.SubnetType.PRIVATE_ISOLATED,
                                    ),
                                    security_groups=[rds_security_group],
                                    credentials=
rds.Credentials.from_generated_secret("Admin"),
                                    instance_type=
ec2.InstanceType.of(ec2.InstanceClass.BURSTABLE3,

ec2.InstanceSize.MICRO),
                                    port= 3306,
                                    allocated_storage= 20,
                                    multi_az= True,
                                    removal_policy= RemovalPolicy.DESTROY,
                                    cloudwatch_logs_exports=['error',
'general', 'slowquery', 'audit'], #comment if db is not deployed
                                    deletion_protection= False,
                                    publicly_accessible= False
                                    )
# Output the RDS endpoint for reference
        CfnOutput(self, 'RDS-Endpoint',
value=myDB.db_instance_endpoint_address)
        CfnOutput(self, "RDSInstanceArn", value=myDB.instance_arn)
```

- ✓ This code creates an RDS database instance named "MyDatabase."
- ✓ The database engine specified is MySQL (rds.DatabaseInstanceEngine.MYSQL).
- ✓ It is deployed within the custom VPC (custom_vpc) and in a private isolated subnet.
- ✓ Credentials for the database are generated using the rds.Credentials.from_generated_secret method with the username "Admin." AWS CDK automatically generates a secure secret in AWS secret manager and associate with RDS instance. The actual username and password are not directly accessible as plain text because they are stored securely under secrets manager.
- ✓ The instance type is set to BURSTABLE3.MICRO (a small, burstable performance instance type).
- ✓ The database listens on port 3306. (default port for MySQL database servers for remote connection)
- ✓ The allocated storage for the database is set to 20 GB.

- ✓ Multi-AZ (Availability Zone) deployment is set to True. A replica (standby replica) of RDS instance will be created in another availability zone in the same region. High availability. Write operation will be happening at the same time in both instances (synchronous operation). But standby replica cannot be accessed directly unless a failure occurs. Backups are taken from the standby replica.
- ✓ The removal policy is set to DESTROY, which means the RDS instance will be deleted when the CDK stack is destroyed.
- ✓ CloudWatch logs are exported for error, general, slow query, and audit logs.
- ✓ Deletion protection is set to False, allowing the RDS instance to be deleted.
- ✓ This code outputs the endpoint and Amazon Resource Name (ARN) of the created RDS instance. This was useful for referencing the RDS instance in other parts of my infrastructure.

SNS topics

- SNS topics is created for RDS.

```
rds_sns_topic = Topic(self, 'RDS SNS topic',display_name='RDS topic')
```

Topic is a construct provided by the CDK for defining SNS topics. Self refers to the instance of the CDK construct or stack where this code resides. 'EKS SNS topic' provides a name for the SNS topic within the CDK construct or stack. display_name='EKS topic': specifies the display name for the SNS topic, which is set to 'EKS topic' in this case.

- Adding Email Subscriptions:

```
eks_sns_topic.add_subscription(EmailSubscription(email_address))
rds_sns_topic.add_subscription(EmailSubscription(email_address))
```

These lines add email subscriptions to both the EKS and RDS SNS topics.
EmailSubscription(email_address): Creates an email subscription for the specified email address.
Email subscriptions allow notifications to be sent to the specified email address when events occur on the associated SNS topics.

- ✓ An SNS topic is a communication channel that allows messages to be sent to multiple subscribers (endpoints) at once.
- ✓ Publishers send messages to a topic.
- ✓ The message can be a simple text message, a JSON object, or other supported formats.
- ✓ Subscribers express interest in receiving messages from a specific topic.
- ✓ Subscribers can be various types of endpoints, such as SQS queues, Lambda functions, HTTP/HTTPS endpoints, email addresses, etc.

- ✓ An SQS queue is a managed message queue service. Messages sent to a queue are stored until a subscriber retrieves and processes them.
- ✓ Subscriptions define how messages from an SNS topic are delivered to endpoints.
- ✓ When an SQS queue is subscribed to an SNS topic, messages published to the topic are delivered to the SQS queue. (For SQS subscriptions, messages are sent to the associated SQS queues.) Each subscriber (SQS queue) receives its own copy of the message.
- ✓ For email subscriptions, notifications are sent to the specified email addresses.

- Cloudwatch integration with RDS

```
# Define CloudWatch alarms for RDS metrics
        rds_cpu_alarm = Alarm(
            self, 'RDSCPUAlarm',
            metric=myDB.metric_cpu_utilization(),
            threshold=40,                    #triggered if the CPU utilization
exceeds 90%.
            evaluation_periods=1,            #how many times the line will be
crossed until alarm rings
            alarm_name='RDSCPUHighAlarm',
            actions_enabled=True
        )

        rds_db_connection_alarm = Alarm(
            self, 'RDSConnectionsEvent',
            metric=myDB.metric_database_connections(),
            threshold=50,                    #triggered if the CPU utilization
exceeds 90%.
            evaluation_periods=1,            #how many times the line will be
crossed until alarm rings
            alarm_name='RDSConnectionsEvent',
            actions_enabled=True
        )


        # Subscribe SNS topics to CloudWatch alarms
        rds_cpu_alarm.add_alarm_action(SnsAction(rds_sns_topic))    #subscribe
sns topic to cloudwatch alarm
        rds_db_connection_alarm.add_alarm_action(SnsAction(rds_sns_topic))

        # Step 4: Configure RDS to Use Log Streams (check AWS RDS documentation)

        # Output the CloudWatch Logs Log Group and Log Stream ARNs
        CfnOutput(self, "LogGroupArn", value=rds_log_group.log_group_name)
        CfnOutput(self, "LogStreamName", value=rds_log_stream.log_stream_name)
```

- ✓ CloudWatch Alarm “rds_cpu_alarm” is created to monitor the CPU utilization of the RDS instance (myDB).

- ✓ It uses the metric_cpu_utilization() method to get the CPU utilization metric for the RDS instance.
- ✓ The alarm is triggered if the CPU utilization exceeds the specified threshold of 40%.
- ✓ evaluation_periods=1 indicates that the threshold must be crossed for at least one consecutive period for the alarm to trigger.
- ✓ alarm_name specifies the name of the CloudWatch Alarm.
- ✓ actions_enabled=True enables actions for the alarm.
- ✓ CloudWatch Alarm "rds_db_connection_alarm" is created to monitor the number of database connections to the RDS instance.
- ✓ It uses the metric_database_connections() method to get the metric for database connections.
- ✓ The alarm is triggered if the number of database connections exceeds the specified threshold of 50.
- ✓ evaluation_periods=1 indicates that the threshold must be crossed for at least one consecutive period for the alarm to trigger.
- ✓ alarm_name specifies the name of the CloudWatch Alarm.
- ✓ actions_enabled=True enables actions for the alarm.

We don't necessarily need AWS lambda to send notifications via SNS from cloudwatch. Cloudwatch has built-in integrations with SNS, allowing us to directly associate SNS topics with cloudwatch. We can use SnsAction class from Aws_cdk.aws_cloudwatch_actions to associate an SNS Topic with alarm.
'rds_cpu_alarm.add_alarm_action' and 'rds_db_connection_alarm.add_alarm_action' add actions to the CloudWatch Alarms. If an alarm is triggered, it will publish a message to the specified SNS topic (rds_sns_topic).

- S3 Bucket creation and exporting cloudwatch logs to s3

```
#S3 bucket creation
        log_export_bucket = s3.Bucket(
            self, 'LogExportBucket' ,
            bucket_name='logexportbucket')

        # Define the bucket policy
        bucket_policy = {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Sid": "uploadlogsmanjitha",
                    "Effect": "Allow",
                    "Principal": {
                        "Service": "logs.eu-central-1.amazonaws.com"
                    },
                    "Action": "s3:GetBucketAcl",
                    "Resource": log_export_bucket.bucket_arn
                },
                {
                    "Sid": "uploadlogs",
```

```
                "Effect": "Allow",
                "Principal": {
                    "Service": "logs.eu-central-1.amazonaws.com"
                },
                "Action": "s3:PutObject",
                "Resource": log_export_bucket.bucket_arn
            }
        ]
    }

    # Attach the bucket policy to the S3 bucket
    s3.BucketPolicy(
        self, 'LogsBucketPolicy',
        bucket=log_export_bucket,
        policy_document=aws_iam.PolicyDocument.from_json(bucket_policy)
    )
```

- Creation of lambda function

```
export_logs_lambda = lambda_.Function(

            self,
            "LogsLambdaFunction",
            handler="lambda_handler.handler",
            runtime=lambda_.Runtime.PYTHON_3_8,
            code=lambda_.Code.from_asset("lambda"),
            role=lambda_role
        )

        # Create an Event Rule to trigger the Lambda function daily
        event_rule = events.Rule(
            self, 'ExportLogsEventRule',
            schedule=events.Schedule.cron(hour='14', minute='35')
        )

        event_rule.add_target(targets.LambdaFunction(export_logs_lambda))
```

The content of Lambda_handeler.py

```
import boto3
import os
import datetime

GROUP_NAME = "/aws/rds/instance/awscaasprojectstack-mydatabase1e2517db-ljeqsyoosnfl/error"
DESTINATION_BUCKET = "logsbucketmanjitha" #change this according to the created bucket name. I
used this logsbucketmanjitha for testing
PREFIX = "Cloudwatchlogstest"
NDAYS = 1
nDays = int(NDAYS)


currentTime = datetime.datetime.now()
StartDate = currentTime - datetime.timedelta(days=nDays)
EndDate = currentTime - datetime.timedelta(days=nDays - 1)


fromDate = int(StartDate.timestamp() * 1000)
toDate = int(EndDate.timestamp() * 1000)

BUCKET_PREFIX = os.path.join(PREFIX, StartDate.strftime('%Y{0}%m{0}%d').format(os.path.sep))
```

```
def lambda_handler(event, context):
    client = boto3.client('logs')
    response = client.create_export_task(
         logGroupName=GROUP_NAME,
         fromTime=fromDate,
         to=toDate,
         destination=DESTINATION_BUCKET,
         destinationPrefix=BUCKET_PREFIX
        )
    print(response)
```

This code creates an AWS Lambda function that exports logs from CloudWatch to an S3 bucket on a daily schedule. It creates an IAM role (LambdaExecutionRole) for the Lambda function. It attaches managed policies granting necessary permissions for basic Lambda execution, Lambda-specific roles, full access to Amazon S3, and full access to CloudWatch.

Creates an Event Rule (ExportLogsEventRule) with a cron schedule to trigger the Lambda function daily at 14:35. Adds the Lambda function (export_logs_lambda) as a target for the event rule, so it gets triggered according to the specified schedule.