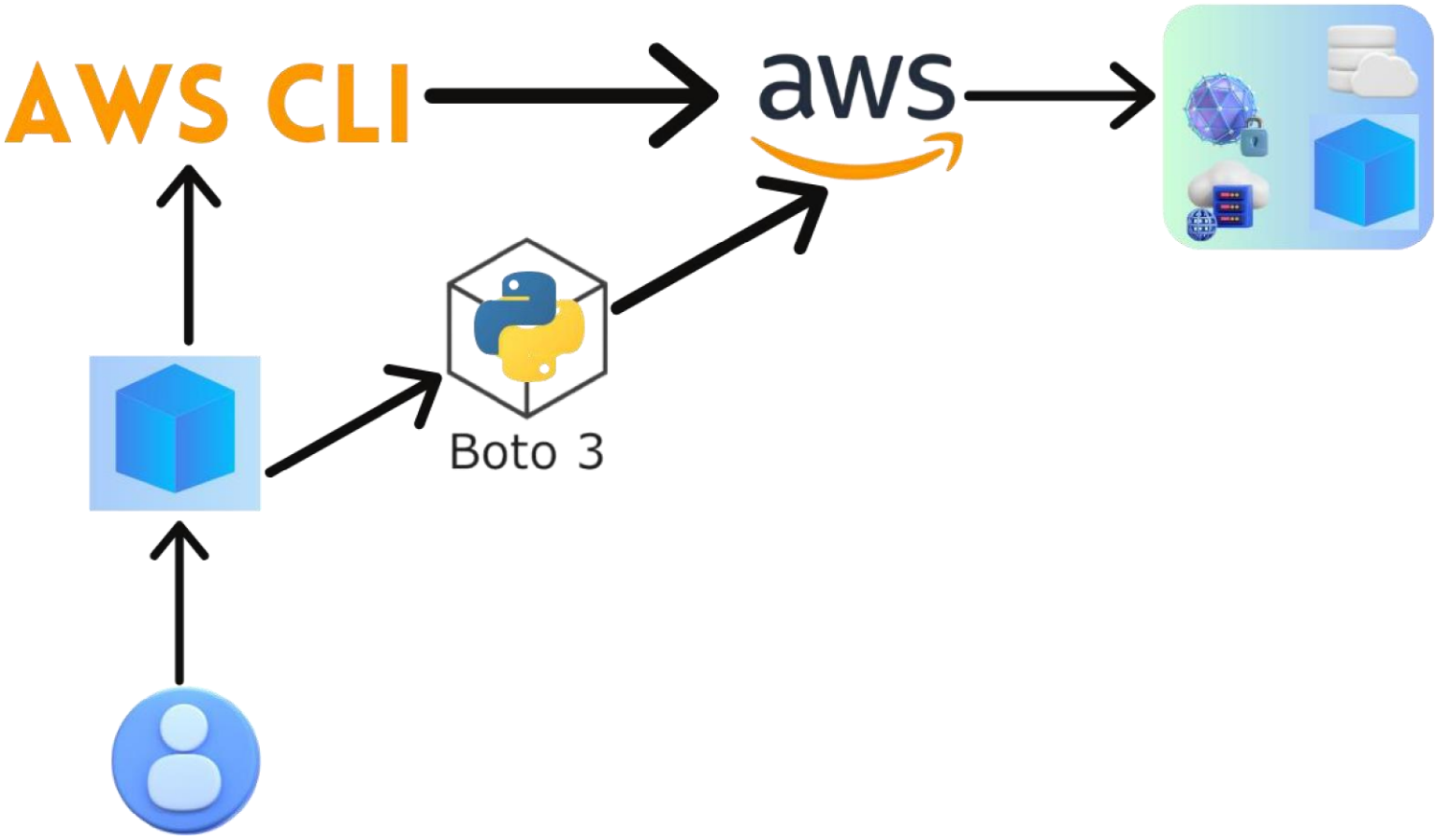
DevOps Shack

Creating & Managing AWS Resources With Python



What is AWS SDK?

The AWS SDK (Software Development Kit) is a collection of tools that allows developers to interact programmatically with AWS services. It provides a set of APIs that enable you to create, configure, and manage AWS services such as EC2, S3, Lambda, and more. The SDK is available in multiple programming languages, including Python, Java, JavaScript, C#, and Ruby, making it versatile and accessible for various development environments.

Key Features of AWS SDK:

1. **Multi-Language Support:** Available for various programming languages.
2. **Simplified API Calls:** Facilitates easy interaction with AWS services.
3. **Robust Documentation:** Comprehensive guides and API references.
4. **Security and Authentication:** Integrated with AWS Identity and Access Management (IAM) for secure API requests.
5. **High-Level Abstractions**: Provides higher-level objects and methods for common tasks.

What is Boto3?

Boto3 is the AWS SDK for Python. It allows Python developers to write software that makes use of services like Amazon S3 and Amazon EC2. Boto3 simplifies the process of interacting with AWS by providing an object-oriented API as well as low-level access to AWS services.

Key Features of Boto3:

1. **Resource API:** Provides an object-oriented API to interact with AWS resources.
2. **Client API:** Allows low-level access to AWS services.
3. **Session Management:** Manages multiple sessions and credentials.
4. **Convenient Pagination:** Automatically handles paginated responses from AWS services.
5. **Waiters:** Simplifies waiting for resource states (e.g., waiting for an EC2 instance to reach the 'running' state).

How Boto3 Works

Boto3 provides two levels of APIs:

1. **Resource API:** Offers an abstraction layer over the raw AWS service APIs, making it easier to work with AWS services. Resources represent an AWS service's objects and their attributes.

- **Example: ec2.Instance('i-1234567890abcdef0') to represent an EC2 instance.**

2. **Client API:** Provides direct access to the AWS service APIs, allowing for more granular control and lower-level interactions.
   - **Example: client.describe_instances(InstanceIds=['i-1234567890abcdef0']) to describe an EC2 instance.**

Steps to Create EC2 Machine & Infra with Python Script:

1. **Ensure AWS CLI is configured**:

aws configure

2. **Create a Virtual Environment and Install Boto3**:

python3 -m venv myenv

source myenv/bin/activate

pip install boto3

3. **Save the Script**: Save the above script as create_ec2.py.

4. **Run the Script**: python create_ec2.py **Script To Create The Resource** *import boto3*

 # Initialize a session using Amazon EC2 ec2 =

boto3.resource('ec2') client = boto3.client('ec2')

 # Create a VPC

vpc = ec2.create_vpc(CidrBlock='10.0.0.0/16') vpc.wait_until_available() print(f'Created VPC: {vpc.id}')

 # Enable DNS support

vpc.modify_attribute(EnableDnsSupport={'Value': True})

vpc.modify_attribute(EnableDnsHostnames={'Value': True}) #

Create an Internet Gateway and attach it to the VPC

internet_gateway = ec2.create_internet_gateway()
vpc.attach_internet_gateway(InternetGatewayId=internet_gateway.id) print(f'Created Internet Gateway: {internet_gateway.id}')

 # Create a public subnet

public_subnet = ec2.create_subnet(CidrBlock='10.0.1.0/24', VpcId=vpc.id, AvailabilityZone='ap-south-1a') print(f'Created Public Subnet: {public_subnet.id}')

 # Create a route table and a public route

route_table = vpc.create_route_table() route = route_table.create_route(DestinationCidrBlock='0.0.0.0/0', GatewayId=internet_gateway.id) print(f'Created Route Table: {route_table.id}')

 # Associate the route table with the subnet

route_table.associate_with_subnet(SubnetId=public_subnet.id) print(f'Associated Route Table with Subnet: {public_subnet.id}')

 # Create a security group

security_group = ec2.create_security_group(GroupName='my-security-group', Description='My security group', VpcId=vpc.id) security_group.authorize_ingress(IpPermissions=[{'IpProtocol': 'tcp', 'FromPort': 22, 'ToPort': 22, 'IpRanges': [{'CidrIp': '0.0.0.0/0'}]}, {'IpProtocol': 'tcp', 'FromPort': 80, 'ToPort': 80, 'IpRanges': [{'CidrIp': '0.0.0.0/0'}]}, {'IpProtocol': 'tcp', 'FromPort': 443, 'ToPort': 443, 'IpRanges': [{'CidrIp': '0.0.0.0/0'}]}, {'IpProtocol': 'tcp', 'FromPort': 3000, 'ToPort': 10000, 'IpRanges': [{'CidrIp': '0.0.0.0/0'}]},]) print(f'Created Security Group: {security_group.id}')

 # Function to create instances

def create_instances(instance_count): instances = ec2.create_instances(ImageId='ami0ad21ae1d0696ad58', # Replace with your desired AMI ID MinCount=instance_count, MaxCount=instance_count, InstanceType='t2.medium', # Choose your instance type KeyName='DevOps', # Replace with your key pair name NetworkInterfaces=[{ 'SubnetId': public_subnet.id, 'DeviceIndex': 0, 'AssociatePublicIpAddress': True, 'Groups': [security_group.group_id] }], TagSpecifications=[{ 'ResourceType': 'instance', 'Tags': [{'Key': 'Name', 'Value': f'Server {i+1}'} for i in range(instance_count)] }], Placement={ 'AvailabilityZone': 'ap-south-1a' }) return instances

 # Number of instances to create num_instances = 1 # Create the instances instances =

create_instances(num_instances) print(f'Created {num_instances} EC2 instances')

 # Print details of the created instances

for instance in instances: print(f'Instance ID: {instance.id}') print(f'State: {instance.state}') print(f'Public DNS: {instance.public_dns_name}') print(f'Instance Type: {instance.instance_type}')

Explaination

The provided script is a comprehensive example of how to automate the creation of AWS infrastructure using Boto3, the AWS SDK for Python. Here's a step-by-step explanation of what each part of the script does:

1. **Initialize a Session Using Amazon EC2**:

   ec2 = boto3.resource('ec2')

   client = boto3.client('ec2')

This initializes the Boto3 resource and client for interacting with the EC2 service.

2. **Create a VPC (Virtual Private Cloud)**:

   vpc = ec2.create_vpc(CidrBlock='10.0.0.0/16')

   vpc.wait_until_available()

   print(f'Created VPC: {vpc.id}')

This creates a VPC with a CIDR block of 10.0.0.0/16 and waits until the VPC is available before proceeding.

3. **Enable DNS Support for the VPC**:

   vpc.modify_attribute(EnableDnsSupport={'Value': True})

vpc.modify_attribute(EnableDnsHostnames={'Value': True})

This enables DNS resolution and DNS hostnames for the VPC.

4. **Create and Attach an Internet Gateway**:

   internet_gateway = ec2.create_internet_gateway()

   vpc.attach_internet_gateway(InternetGatewayId=internet_gateway.id)

   print(f'Created Internet Gateway: {internet_gateway.id}')

This creates an Internet Gateway and attaches it to the VPC, enabling the VPC to communicate with the internet.

5. **Create a Public Subnet**:

   public_subnet = ec2.create_subnet(CidrBlock='10.0.1.0/24', VpcId=vpc.id, AvailabilityZone='ap-south-1a')

   print(f'Created Public Subnet: {public_subnet.id}')

This creates a public subnet within the VPC in the specified Availability Zone (ap-south-1a).

6. **Create a Route Table and a Public Route**: *route_table = vpc.create_route_table()*

   route = route_table.create_route(DestinationCidrBlock='0.0.0.0/0',

```
    GatewayId=internet_gateway.id
)
print(f'Created Route Table: {route_table.id}')
```

This creates a route table and adds a route to the internet via the Internet Gateway.

7. **Associate the Route Table with the Subnet**:

```
route_table.associate_with_subnet(SubnetId=public_subnet.id)
print(f'Associated Route Table with Subnet: {public_subnet.id}')
```

This associates the route table with the public subnet, making the subnet public.

8. **Create a Security Group**:

```
security_group = ec2.create_security_group(
    GroupName='my-security-group',
    Description='My security group',
    VpcId=vpc.id
)
security_group.authorize_ingress(
    IpPermissions=[
        {'IpProtocol': 'tcp', 'FromPort': 22, 'ToPort': 22, 'IpRanges': [{'CidrIp': '0.0.0.0/0'}]},
{'IpProtocol': 'tcp', 'FromPort': 80, 'ToPort': 80, 'IpRanges': [{'CidrIp': '0.0.0.0/0'}]},
        {'IpProtocol': 'tcp', 'FromPort': 443, 'ToPort': 443, 'IpRanges': [{'CidrIp':
'0.0.0.0/0'}]},
        {'IpProtocol': 'tcp', 'FromPort': 3000, 'ToPort': 10000, 'IpRanges': [{'CidrIp':
'0.0.0.0/0'}]},
    ]
)
print(f'Created Security Group: {security_group.id}')
```

This creates a security group within the VPC and authorizes ingress (inbound traffic) on ports 22 (SSH), 80 (HTTP), 443 (HTTPS), and a range from 3000 to 10000 (custom applications).

9. **Function to Create EC2 Instances**:

```
def create_instances(instance_count):
instances = ec2.create_instances(
```

```
        ImageId='ami-0ad21ae1d0696ad58',  # Replace with your desired AMI ID
        MinCount=instance_count,
        MaxCount=instance_count,
        InstanceType='t2.medium',  # Choose your instance type
        KeyName='DevOps',  # Replace with your key pair name
        NetworkInterfaces=[{
            'SubnetId': public_subnet.id,
            'DeviceIndex': 0,
            'AssociatePublicIpAddress': True,
          'Groups': [security_group.group_id]
        }],
        TagSpecifications=[
            {
                'ResourceType': 'instance',
                'Tags': [{'Key': 'Name', 'Value': f'Server {i+1}'} for i in range(instance_count)]
            }
        ],
        Placement={
            'AvailabilityZone': 'ap-south-1a'
        }
    )
    return instances
```

This function creates EC2 instances with the specified configuration: AMI ID, instance type, key pair, network interface, tags, and placement in the specified Availability Zone.

10. **Create the Instances**:
```
num_instances = 1
instances = create_instances(num_instances)
print(f'Created {num_instances} EC2 instances')
```

for instance in instances:

print(f'Instance ID: {instance.id}')

print(f'State: {instance.state}')

print(f'Public DNS: {instance.public_dns_name}')

print(f'Instance Type: {instance.instance_type}')

This part calls the create_instances function to create the specified number of instances (in this case, 1) and prints details of the created instances.

The script automates the entire process of setting up a VPC with all necessary components and launching an EC2 instance within it. It also ensures proper tagging and placement of resources for better management and organization.

Script To Destroy the Resources

The error indicates that the route table still has dependencies, which usually means it is still associated with one or more subnets or has active routes that need to be deleted first. To resolve this, you need to:

1. Disassociate the route table from the subnet.
2. Delete any custom routes in the route table before deleting the route table itself.

Here's the updated script to handle these dependencies:

import boto3

 # Initialize a session using Amazon EC2

ec2 = boto3.resource('ec2') client =

boto3.client('ec2')

 # Replace with the IDs of the resources created by your script

vpc_id = 'vpc-06ca907983f09ea67' internet_gateway_id =

'igw-0d46bf5e38f6ed16b' subnet_id = 'subnet-

0664dd2777d5cf809' route_table_id = 'rtb-

00ac54440f0998086' security_group_id = 'sg-

0a681424409388ea3'

```
# Terminate all instances within the VPC print(f'Terminating instances in
VPC: {vpc_id}') instances = ec2.instances.filter(Filters=[{'Name': 'vpc-id',
'Values': [vpc_id]}]) for instance in instances:     instance.terminate()
instance.wait_until_terminated()     print(f'Terminated instance:
{instance.id}')

# Detach and delete the internet gateway print(f'Detaching and deleting
Internet Gateway: {internet_gateway_id}') vpc = ec2.Vpc(vpc_id)
vpc.detach_internet_gateway(InternetGatewayId=internet_gateway_id)
internet_gateway = ec2.InternetGateway(internet_gateway_id)
internet_gateway.delete() print(f'Deleted Internet Gateway:
{internet_gateway_id}')

# Disassociate the route table from the subnet
print(f'Disassociating Route Table from Subnet: {subnet_id}')
route_table = ec2.RouteTable(route_table_id) associations =
route_table.associations for association in associations:
    association.delete()     print(f'Disassociated Route Table: {route_table_id}
from Subnet: {subnet_id}')

# Delete custom routes in the route table print(f'Deleting
custom routes in Route Table: {route_table_id}') for route in
route_table.routes:
    if route.destination_cidr_block != 'local':  # Skip the default local route
client.delete_route(RouteTableId=route_table_id,
DestinationCidrBlock=route.destination_cidr_block)
        print(f'Deleted route: {route.destination_cidr_block}')
```

```
# Delete the route table print(f'Deleting Route
Table: {route_table_id}') route_table.delete()
print(f'Deleted Route Table: {route_table_id}')

# Delete the security group print(f'Deleting Security
Group: {security_group_id}') security_group =
ec2.SecurityGroup(security_group_id)
security_group.delete() print(f'Deleted Security Group:
{security_group_id}')

# Delete the subnet print(f'Deleting
Subnet: {subnet_id}') subnet =
ec2.Subnet(subnet_id)
subnet.delete() print(f'Deleted
Subnet: {subnet_id}')

# Finally, delete the VPC
print(f'Deleting VPC: {vpc_id}')
vpc.delete() print(f'Deleted
VPC: {vpc_id}')
```

Explanation of the Updated Script

1. **Initialize a Session Using Amazon EC2**:

   ```
   import boto3
   ec2 = boto3.resource('ec2')
   client = boto3.client('ec2')
   ```

This initializes the Boto3 resource and client for interacting with the EC2 service.

2. **Define Resource IDs**:

vpc_id = 'vpc-06ca907983f09ea67'

internet_gateway_id = 'igw-0d46bf5e38f6ed16b'

subnet_id = 'subnet-0664dd2777d5cf809'

route_table_id = 'rtb-00ac54440f0998086'

security_group_id = 'sg-0a681424409388ea3'

These variables store the IDs of the resources created by your script.

3. **Terminate All Instances Within the VPC**:

   print(f'Terminating instances in VPC: {vpc_id}') instances =

   ec2.instances.filter(Filters=[{'Name': 'vpc-id', 'Values': [vpc_id]}]) for

   instance in instances: instance.terminate()

   instance.wait_until_terminated()

   print(f'Terminated instance: {instance.id}')

This part filters all instances within the specified VPC and terminates them. It waits until each instance is fully terminated before proceeding.

4. **Detach and Delete the Internet Gateway**: *print(f'Detaching and deleting Internet*

   Gateway: {internet_gateway_id}') vpc = ec2.Vpc(vpc_id)

   vpc.detach_internet_gateway(InternetGatewayId=internet_gateway_id)

   internet_gateway = ec2.InternetGateway(internet_gateway_id)

   internet_gateway.delete() print(f'Deleted Internet

Gateway: {internet_gateway_id}') This detaches the internet

gateway from the VPC and then deletes it.

5. **Disassociate the Route Table from the Subnet**: *print(f'Disassociating Route Table*

   from Subnet: {subnet_id}') route_table = ec2.RouteTable(route_table_id)

   associations = route_table.associations for

   association in associations:

   association.delete() print(f'Disassociated Route Table:

{route_table_id} from Subnet: {subnet_id}')

This disassociates the route table from any subnets. The route table can have multiple associations, and this ensures all are removed.

6. **Delete Custom Routes in the Route Table**:

   print(f'Deleting custom routes in Route Table: {route_table_id}') for

   route in route_table.routes:

   if route.destination_cidr_block != 'local': # Skip the default local route

   client.delete_route(RouteTableId=route_table_id, DestinationCidrBlock=route.destination_cidr_block)

print(f'Deleted route: {route.destination_cidr_block}')

This deletes any custom routes in the route table, except for the default local route.

7. **Delete the Route Table**:

   print(f'Deleting Route Table: {route_table_id}')

   route_table.delete() print(f'Deleted

Route Table: {route_table_id}')

After disassociating and deleting custom routes, the route table can be deleted.

8. **Delete the Security Group**:

   print(f'Deleting Security Group: {security_group_id}')

   security_group = ec2.SecurityGroup(security_group_id)

   security_group.delete() print(f'Deleted

Security Group: {security_group_id}')

This deletes the security group associated with the VPC.

9. **Delete the Subnet**:

   print(f'Deleting Subnet: {subnet_id}')

   subnet = ec2.Subnet(subnet_id)

   subnet.delete()

   print(f'Deleted Subnet: {subnet_id}')

This deletes the subnet.

10. **Finally, Delete the VPC**:

    print(f'Deleting VPC: {vpc_id}')

    vpc.delete()

print(f'Deleted VPC: {vpc_id}')

After all other resources are deleted, the VPC itself can be deleted.