AWS Service Configuration with UI, CLI, CloudFormation, Terraform, Pulumi- (Part-1)

Compute

1. **EC2 (Elastic Compute Cloud)**: Scalable virtual servers for running applications.
2. **Lambda**: Serverless computing service to run code without provisioning servers.
3. **ECS (Elastic Container Service)**: Container orchestration for Docker containers.
4. **EKS (Elastic Kubernetes Service)**: Managed Kubernetes service for containerized applications.
5. **Fargate**: Serverless compute engine for containers.
6. **Batch**: Fully managed batch processing at scale.
7. **Elastic Beanstalk**: Platform-as-a-Service (PaaS) for deploying and managing applications.
8. **Outposts**: AWS infrastructure extension to on-premises locations.
9. **Lightsail**: Simplified cloud computing for small businesses and developers.

Storage

1. **S3 (Simple Storage Service)**: Object storage.
2. **EBS (Elastic Block Store)**: Block storage for EC2.
3. **EFS (Elastic File System)**: Managed file storage.
4. **FSx**: Managed file systems (Windows, Lustre, NetApp).
5. **Glacier**: Low-cost archival storage.
6. **Storage Gateway**: Hybrid cloud storage.

Database

1. **RDS (Relational Database Service)**: Managed relational databases.
2. **DynamoDB**: NoSQL database.
3. **Redshift**: Data warehousing.
4. **ElastiCache**: In-memory caching (Redis/Memcached).
5. **Neptune**: Graph database.
6. **DocumentDB**: Managed NoSQL document database.
7. **Timestream**: Time-series database.
8. **Keyspaces**: Managed Cassandra-compatible database.

Networking & Content Delivery

1. **VPC (Virtual Private Cloud)**: Isolated network for AWS resources.
2. **CloudFront**: Content delivery network (CDN).
3. **Route 53**: Domain Name System (DNS) service.
4. **Direct Connect**: Dedicated network connection to AWS.
5. **Global Accelerator**: Improves application performance.
6. **Elastic Load Balancing (ELB)**: Distributes incoming traffic.
7. **App Mesh**: Service mesh for microservices.

Developer Tools

1. **CodeCommit**: Version control (Git).
2. **CodeBuild**: Build and test code.
3. **CodeDeploy**: Automate code deployments.
4. **CodePipeline**: Continuous integration and delivery (CI/CD).
5. **Cloud9**: Online IDE.
6. **X-Ray**: Debugging and tracing applications.

Security, Identity, & Compliance

1. **IAM (Identity and Access Management)**: Manage user permissions.
2. **Cognito**: User authentication and identity.
3. **Secrets Manager**: Securely manage secrets.
4. **Certificate Manager**: Manage SSL/TLS certificates.
5. **GuardDuty**: Threat detection.

6. **Inspector**: Security assessment.
7. **Macie**: Data security and privacy.
8. **Security Hub**: Centralized security view.
9. **WAF (Web Application Firewall)**: Protect web applications.
10. **Shield**: DDoS protection.

Analytics

1. **Athena**: Query data in S3 using SQL.
2. **Glue**: ETL (Extract, Transform, Load) service.
3. **QuickSight**: Business intelligence and data visualization.
4. **EMR (Elastic MapReduce)**: Big data processing.
5. **Kinesis**: Real-time data streaming.
6. **Data Pipeline**: Data workflow orchestration.
7. **OpenSearch Service**: Search and analytics.

Machine Learning

1. **SageMaker**: Build, train, and deploy ML models.
2. **Rekognition**: Image and video analysis.
3. **Comprehend**: Natural language processing.
4. **Polly**: Text-to-speech.
5. **Translate**: Language translation.
6. **Lex**: Build conversational interfaces.
7. **Personalize**: Personalization and recommendations.
8. **Forecast**: Time-series forecasting.

Management & Governance

1. **CloudWatch**: Monitoring and logging.
2. **CloudFormation**: Infrastructure as code.
3. **CloudTrail**: Governance and compliance.
4. **Config**: Resource compliance tracking.
5. **Trusted Advisor**: Resource optimization and recommendations.
6. **Control Tower**: Multi-account governance.
7. **Organizations**: Manage multiple AWS accounts.

8. **Service Catalog**: Centralized application management.

Migration & Transfer

1. **DMS (Database Migration Service)**: Migrate databases to AWS.
2. **Migration Hub**: Centralized migration tracking.
3. **Snowball/Snowcone/Snowmobile**: Physical data transfer.
4. **DataSync**: Data transfer automation.

Media Services

1. **MediaConvert**: File-based video processing.
2. **MediaLive**: Live video processing.
3. **MediaPackage**: Video packaging and delivery.

Installation of AWS CLI, Terraform & Pulumi

1. AWS Command Line Interface(CLI) Installation

Installation on Ubuntu/Linux

curl "https://awscli.amazonaws.com/AWSCLIV2.pkg" -o "awscliv2.pkg"

sudo installer -pkg awscliv2.pkg -target /

Installation on Windows

1. Download the installer from AWS CLI Download
2. Run the .msi installer and follow the setup instructions.

Verify Installation

aws --version

---

2. Terraform Installation

Installation on Ubuntu/Linux

sudo apt update && sudo apt install -y gnupg software-properties-common

wget -O- https://apt.releases.hashicorp.com/gpg | gpg --dearmor | sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg

echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list

sudo apt update && sudo apt install terraform

Installation on Windows

1. Download the Terraform binary from Terraform Download
2. Extract and add Terraform to the system PATH.

Verify Installation

terraform version

---

3. Pulumi Installation

1. Install Pulumi and Set Up Project

On Linux or macOS:

1. Open your terminal.
2. Run the following command to install Pulumi using the package manager for your system:

For macOS (Homebrew):
brew install pulumi

For Linux (APT):
curl -fsSL https://get.pulumi.com | sh

For Linux (YUM):
curl -fsSL https://get.pulumi.com | sudo sh

This will download and install the Pulumi CLI on your system.

After installation, verify it by running:
pulumi version

On Windows:

1. Download the latest release from the Pulumi GitHub releases page.
2. Extract the downloaded archive.
3. Add the folder where pulumi.exe is located to your system's PATH.

Verify the installation by running:
pulumi version

Alternative: Using Package Managers:

For Windows (via Chocolatey):
choco install pulumi

For macOS (via Homebrew):
brew install pulumi

Compute

1. EC2 (Elastic Compute Cloud): Scalable virtual servers for running applications.

EC2 Introduction: Amazon EC2 (Elastic Compute Cloud) offers scalable virtual servers in the cloud for running applications. It provides flexibility to choose instance types, operating systems, and configurations to suit specific workloads. EC2 enables high availability and scalability, allowing users to adjust resources based on demand.

---

1. Configuration with UI

Configuration Steps:

- **Sign in to AWS Console:** Log in to your AWS account and open the EC2 dashboard.
- **Launch an Instance:**
  - Click "Launch Instance."
  - Select an Amazon Machine Image (AMI) (e.g., Amazon Linux, Ubuntu, Windows).
  - Choose an instance type (e.g., t2.micro for a basic setup).
- **Configure Instance:**
  - Set the number of instances and configure network settings (VPC and subnet).
  - Choose IAM role, monitoring, and tenancy options.
- **Add Storage:**
  - Add EBS volume (Elastic Block Store) for persistent storage.
- **Add Tags:**
  - Create tags to organize and identify your instance (e.g., Name: MyServer).
- **Configure Security Group:**
  - Set up firewall rules (e.g., open port 22 for SSH, port 80 for HTTP).

- **Review and Launch:**
  - Review all settings and click “Launch.”
  - Select or create a new SSH key pair for secure access.
- **Access Your Instance:**
  - Once the instance is running, connect via SSH (for Linux) or RDP (for Windows) using the instance’s public IP.

---

2. Launch an EC2 Instance (via AWS CLI)

aws ec2 run-instances --image-id <ami-id> --count 1 --instance-type <instance-type> --key-name <key-pair-name> --security-group-ids <security-group-id> --subnet-id <subnet-id> --tag-specifications 'ResourceType=instance,Tags=[{Key=Name,Value=MyInstance}]'

- Replace <ami-id> with the AMI ID (e.g., ami-0c55b159cbfafe1f0 for Amazon Linux).
- Replace <instance-type> with your desired instance type (e.g., t2.micro).
- Replace <key-pair-name> with your SSH key pair name.
- Replace <security-group-id> and <subnet-id> with appropriate security group and subnet IDs.

2. Connect to Your EC2 Instance (via SSH)

Once the instance is running, connect to it using SSH (for Linux instances):

ssh -i <path-to-your-key-pair.pem> ec2-user@<public-ip>

- Replace <path-to-your-key-pair.pem> with the path to your .pem key pair file.
- Replace <public-ip> with your instance’s public IP address.

3. Stop the EC2 Instance

aws ec2 stop-instances --instance-ids <instance-id>

- Replace <instance-id> with the instance ID of the EC2 instance you want to stop.

4. Start the EC2 Instance

aws ec2 start-instances --instance-ids <instance-id>

5. Terminate the EC2 Instance

aws ec2 terminate-instances --instance-ids <instance-id>

6.To check the status of your EC2 instance:

aws ec2 describe-instances --instance-ids <instance-id>

---

3. CloudFormation template Configuration (in YAML format):

1. CloudFormation Template (YAML)

yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Simple EC2 instance setup with CloudFormation
Resources:
  MyEC2Instance:
    Type: AWS::EC2::Instance
    Properties:
      InstanceType: t2.micro
      KeyName: <your-key-pair-name>
      ImageId: <ami-id>
      SecurityGroups:
```

```
      - <security-group-id>
    Tags:
      - Key: Name
        Value: MyEC2Instance
    AvailabilityZone: <availability-zone>
Outputs:
  InstanceId:
    Description: "Instance ID"
    Value: !Ref MyEC2Instance
  PublicIP:
    Description: "Public IP address"
    Value: !GetAtt MyEC2Instance.PublicIp
```

Explanation:

- **InstanceType:** Defines the type of instance (e.g., t2.micro).
- **KeyName:** Your SSH key pair name to access the instance.
- **ImageId:** The AMI ID to use (e.g., ami-0c55b159cbfafe1f0 for Amazon Linux).
- **SecurityGroups:** The security group ID to associate with the instance.
- **AvailabilityZone:** The availability zone where the instance will be launched (optional).
- **Tags:** Tags applied to the instance, like setting the name.
- **Outputs:** Outputs the instance ID and public IP after creation.

2. Deploy the CloudFormation Template

After creating the CloudFormation template (let's save it as ec2-template.yaml), you can deploy it using the AWS CLI:

aws cloudformation create-stack --stack-name MyEC2Stack --template-body file://ec2-template.yaml

- **--stack-name:** The name you want to give to your CloudFormation stack (e.g., MyEC2Stack).
- **--template-body:** The path to your CloudFormation template.

3. Check the Stack Creation Status

To monitor the creation of the stack, use:

aws cloudformation describe-stacks --stack-name MyEC2Stack

This will provide details of the stack and show if the EC2 instance has been successfully launched.

4. Access Your EC2 Instance

Once the stack is created, retrieve the public IP address of the EC2 instance using the following command:

aws cloudformation describe-stacks --stack-name MyEC2Stack --query "Stacks[0].Outputs"

This will return the instance's public IP, which you can use to SSH into the instance (as shown earlier):

ssh -i <path-to-your-key-pair.pem> ec2-user@<public-ip>

5. Update the Stack (If Necessary)

If you need to modify the configuration, you can update the CloudFormation stack by providing the updated template:

aws cloudformation update-stack --stack-name MyEC2Stack --template-body file://ec2-template.yaml

6. Delete the Stack (When Done)

aws cloudformation delete-stack --stack-name MyEC2Stack

This will clean up the resources created by the CloudFormation stack.

---

4. Terraform Configuration

To configure and deploy an EC2 instance using Terraform, you will write a Terraform configuration file to define the infrastructure as code. Below is an example of how to create an EC2 instance using Terraform.

1. Terraform Configuration (main.tf)

hcl

```
provider "aws" {
  region = "us-east-1"  # Choose your AWS region
}

resource "aws_instance" "my_ec2" {
  ami           = "<ami-id>"         # Specify the AMI ID
  instance_type = "t2.micro"         # Choose the instance type
  key_name      = "<key-pair-name>"  # Specify the key pair name

  # Optional: Add security groups (can be created or referenced by name)
  vpc_security_group_ids = ["<security-group-id>"]

  tags = {
    Name = "MyEC2Instance"
  }

  # Optional: Set user data (e.g., install a script or configure instance on launch)
  user_data = <<-EOF
```

```
            #!/bin/
            echo "Hello, World" > /home/ec2-user/hello.txt
            EOF
}

# Optional: Output the instance ID and public IP
output "instance_id" {
  value = aws_instance.my_ec2.id
}

output "public_ip" {
  value = aws_instance.my_ec2.public_ip
}
```

Explanation:

- **provider "aws"**: Specifies the AWS provider and region. You can adjust this for your AWS region.
- **aws_instance**: Defines the EC2 instance resource.
    - ami: The AMI ID you want to use (e.g., ami-0c55b159cbfafe1f0 for Amazon Linux).
    - instance_type: The type of instance you want (e.g., t2.micro).
    - key_name: Your SSH key pair to allow SSH access.
    - vpc_security_group_ids: Security group IDs that allow inbound traffic (e.g., port 22 for SSH).
    - **tags**: Adds tags for organizing resources.
    - **user_data**: Optional. You can pass a shell script to configure the instance when it launches.

2. Initialize Terraform

Before you apply the configuration, initialize your Terraform working directory:

```
terraform init
```

This command initializes the working directory containing your Terraform configuration files and downloads the necessary provider plugins.

3. Plan the Terraform Deployment

terraform plan

This shows a preview of the actions Terraform will perform, like creating the EC2 instance.

4. Apply the Configuration

To create the EC2 instance and other resources, run:

terraform apply

You will be prompted to confirm the execution. Type yes to proceed. Terraform will then create the EC2 instance and any associated resources based on the configuration.

5. Access Your EC2 Instance

After the EC2 instance is launched, retrieve the public IP address using the output:

terraform output public_ip

SSH into the instance using:

ssh -i <path-to-your-key-pair.pem> ec2-user@<public-ip>

6. Destroy the EC2 Instance

Once you're done, you can destroy the resources created by Terraform to clean up:

terraform destroy

---

5. Configuration with Pulumi:

Create a new Pulumi project for AWS in your desired language (for this example, we'll use TypeScript):

pulumi new aws-typescript

This will guide you through creating a new project with a basic setup.

2. Pulumi Code (index.ts)

In the index.ts file, add the following code to provision an EC2 instance using Pulumi:

typescript

```
import * as aws from "@pulumi/aws";
import * as pulumi from "@pulumi/pulumi";

// Define the region for the EC2 instance
const region = "us-east-1"; // Set your preferred region

// Create a security group to allow SSH access
const securityGroup = new aws.ec2.SecurityGroup("my-ec2-sg", {
    ingress: [
        {
            protocol: "tcp",
            fromPort: 22,
            toPort: 22,
            cidrBlocks: ["0.0.0.0/0"], // Allow SSH from anywhere
        },
    ],
    egress: [
```

```
        {
            protocol: "-1", // Allow all outbound traffic
            fromPort: 0,
            toPort: 0,
            cidrBlocks: ["0.0.0.0/0"],
        },
    ],
});
```

// Create an EC2 instance

```
const myInstance = new aws.ec2.Instance("my-ec2-instance", {
    ami: "<ami-id>", // Replace with your AMI ID (e.g., Amazon Linux)
    instanceType: "t2.micro", // Choose your instance type
    keyName: "<key-pair-name>", // Replace with your SSH key pair name
    securityGroups: [securityGroup.name],
    tags: {
        Name: "MyEC2Instance",
    },
});

// Export the instance ID and public IP
export const instanceId = myInstance.id;
export const publicIp = myInstance.publicIp;
```

Explanation:

- **aws.ec2.SecurityGroup**: Creates a security group that allows inbound SSH (port 22) from anywhere (0.0.0.0/0).
- **aws.ec2.Instance**: Creates an EC2 instance with the specified ami ID, instance type, and SSH key name.
- **export**: Exports the instance's ID and public IP so you can access them later.

3. Install Dependencies

If you are using TypeScript, make sure you have the necessary dependencies installed:

npm install @pulumi/aws

4. Preview the Deployment

pulumi preview

This will show you the resources that will be created and any changes that will be applied.

5. Deploy the EC2 Instance

To deploy the EC2 instance, run:

pulumi up

6. Access the EC2 Instance

Once the EC2 instance is created, you can retrieve the public IP address:

pulumi stack output publicIp

Then, you can SSH into the EC2 instance using the following command:

ssh -i <path-to-your-key-pair.pem> ec2-user@<public-ip>

7. Destroy the EC2 Instance

When you're done, you can clean up the resources by destroying the stack:

pulumi destroy

Conclusion:

Pulumi allows you to define your infrastructure using programming languages like TypeScript, Python, or Go. It provides the flexibility of infrastructure as code while maintaining the simplicity and power of traditional programming languages. The steps above demonstrate how to use Pulumi to provision an EC2 instance with minimal effort and in a repeatable manner.

---

2. Lambda: Serverless computing service to run code without provisioning servers.

AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS) that lets you run code in response to events without the need to provision or manage servers. You simply upload your code (known as a "Lambda function"), and Lambda automatically handles the scaling, monitoring, and administration of the infrastructure. It can be triggered by various AWS services such as S3, DynamoDB, or API Gateway. Lambda is cost-effective because you only pay for the compute time consumed by your function, not for idle resources. It's ideal for event-driven applications, real-time file processing, and microservices.

---

1. Manual Configuration (AWS Management Console):

- **Step 1**: Go to the AWS Lambda Console.
- **Step 2**: Click on "Create function."
- **Step 3**: Choose "Author from scratch," provide a function name, select the runtime (e.g., Python 3.x), and set the execution role (create a new role or use an existing one).
- **Step 4**: Configure the function with the necessary environment variables, VPC, or other options.
- **Step 5**: Write or upload your Lambda function code.

- **Step 6**: Set up triggers (e.g., API Gateway, S3, DynamoDB) to invoke your function.
- **Step 7**: Click "Create function."

---

2. CLI Configuration (AWS CLI):

Step 1: Create a Lambda function using the AWS CLI.
aws lambda create-function --function-name MyLambdaFunction \
--runtime python3.8 --role arn:aws:iam::account-id:role/service-role/role-name \
--handler lambda_function.lambda_handler --zip-file fileb://function.zip

- Replace MyLambdaFunction, python3.8, and other placeholders with the actual values.

Step 2: Update the function's code if needed.
aws lambda update-function-code --function-name MyLambdaFunction \
--zip-file fileb://function.zip

---

3. CloudFormation Configuration:

You can define your Lambda function and related resources in a CloudFormation template. Example template:

yaml

```
Resources:
  MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
```

```
FunctionName: MyLambdaFunction
Handler: index.handler
Role: arn:aws:iam::account-id:role/service-role/role-name
Code:
  S3Bucket: my-bucket
  S3Key: function.zip
Runtime: nodejs14.x
```

- **Step 1**: Save the CloudFormation template in a file (e.g., lambda-template.yaml).

Step 2: Deploy the stack using AWS CLI:
aws cloudformation create-stack --stack-name my-stack --template-body file://lambda-template.yaml

---

4. Terraform Configuration:

With Terraform, you can define the Lambda function resource in a .tf file.

Example:

hcl

```
resource "aws_lambda_function" "my_lambda" {
  function_name = "MyLambdaFunction"
  role          = "arn:aws:iam::account-id:role/service-role/role-name"
  handler       = "index.handler"
  runtime       = "nodejs14.x"
  filename      = "function.zip"

  source_code_hash = filebase64sha256("function.zip")
}
```

Step 1: Initialize the Terraform working directory:
terraform init

- **Step 2**: Apply the configuration:
  terraform apply

---

5. Configuration with Pulumi:

Step 1: Install Pulumi & Set Up Project:

If you haven't already installed Pulumi, do so by following the installation instructions.

After installation, set up a new Pulumi project:

pulumi new typescript

Step 2: Install Dependencies:

Install the Pulumi AWS SDK:

npm install @pulumi/aws

Step 3: Write the Lambda Configuration:

In the index.ts (or similar entry file), write the Pulumi code to define an AWS Lambda function.

Here's an example of creating a Lambda function using Pulumi in **TypeScript**:

typescript

import * as aws from "@pulumi/aws";

```
import * as pulumi from "@pulumi/pulumi";

// Create an IAM role for the Lambda function
const role = new aws.iam.Role("lambda-role", {
    assumeRolePolicy: JSON.stringify({
        Version: "2012-10-17",
        Statement: [
            {
                Action: "sts:AssumeRole",
                Effect: "Allow",
                Principal: {
                    Service: "lambda.amazonaws.com",
                },
            },
        ],
    }),
});

// Lambda function code (inline example or upload a .zip)
const lambdaFunction = new aws.lambda.Function("myLambdaFunction", {
    role: role.arn,
    handler: "index.handler", // function name in your handler code
    runtime: aws.lambda.NodeJS14dXRuntime,
    code: new aws.s3.BucketObject("function.zip", {
        bucket: "my-lambda-code-bucket",
        source: new pulumi.asset.FileAsset("path/to/your/lambda.zip"),
    }).bucket.apply(bucket => `s3://${bucket}/function.zip`),
});

// Export the Lambda function name and ARN
export const lambdaFunctionName = lambdaFunction.name;
export const lambdaFunctionArn = lambdaFunction.arn;
```

- **IAM Role**: The Lambda function requires a role to execute. The IAM role is created with a trust policy allowing the Lambda service to assume the role.
- **Lambda Code**: This example assumes the Lambda function code is in a .zip file uploaded to an S3 bucket. You can modify this to use inline code or upload code directly as a .zip from your local machine.

Step 4: Run Pulumi Commands:

Login to Pulumi (if not logged in yet):
pulumi login
Initialize the Stack (if you haven't initialized it already):
pulumi stack init dev

Deploy the Infrastructure:
pulumi up

This will prompt you to review the changes that will be made (such as creating a Lambda function, IAM role, and other resources) before confirming and applying them.

Step 5: Verify:

Once the stack is deployed, you can verify the Lambda function has been created from the **AWS Management Console**, or by using AWS CLI:

aws lambda get-function --function-name MyLambdaFunction

Conclusion:

This Pulumi configuration will create an AWS Lambda function, along with the necessary IAM role, and configure it to run a function defined in a .zip file stored in S3.

---

3. ECS (Elastic Container Service): Container orchestration for Docker containers.

Amazon ECS (Elastic Container Service) is a fully managed service that helps you run and scale Docker containers on AWS. It simplifies container orchestration by managing the infrastructure, allowing you to focus on your applications. ECS supports both EC2 and serverless Fargate launch types, providing flexibility and scalability for containerized workloads.

---

1. Configuration (AWS Console)

Steps:

1. **Create a Cluster**:
    - Go to the ECS console.
    - Click on **Create cluster** and choose the type (EC2 or Fargate).
    - Follow the prompts to configure your cluster.
2. **Create a Task Definition**:
    - Navigate to the **Task Definitions** tab.
    - Click **Create new Task Definition**.
    - Choose **Fargate** or **EC2**, depending on your cluster.
    - Define the task with the necessary Docker image and container settings.
3. **Run a Task or Service**:
    - Once the task definition is created, go to **Clusters**, choose your cluster.
    - Click **Create Service** and configure it to run your tasks.
    - Choose **Fargate** or **EC2** as the launch type, configure desired instances, and scale settings.
4. **Configure Networking**:
    - Define the VPC, subnets, and security groups.
    - Attach an Elastic Load Balancer (ELB) if needed for your services.

---

2. CLI Configuration

Steps:

Create a Cluster:
aws ecs create-cluster --cluster-name my-cluster

1. **Create a Task Definition**:

Create a JSON file task-definition.json:
json

```
{
  "family": "my-task",
  "containerDefinitions": [
    {
      "name": "my-container",
      "image": "my-docker-image",
      "memory": 512,
      "cpu": 256,
      "essential": true
    }
  ]
}
```

Register the task definition:
aws ecs register-task-definition --cli-input-json file://task-definition.json

Run a Task:
aws ecs run-task --cluster my-cluster --task-definition my-task --launch-type FARGATE --network-configuration

"awsvpcConfiguration={subnets=[subnet-id],securityGroups=[sg-id],assignPublicIp=ENABLED}"

Create a Service:
aws ecs create-service --cluster my-cluster --service-name my-service --task-definition my-task --desired-count 1 --launch-type FARGATE --network-configuration "awsvpcConfiguration={subnets=[subnet-id],securityGroups=[sg-id],assignPublicIp=ENABLED}"

---

3. CloudFormation Template

A CloudFormation template can automate ECS cluster creation, task definition, and services.

yaml

```
Resources:
  MyCluster:
    Type: AWS::ECS::Cluster
    Properties:
      ClusterName: my-cluster

  MyTaskDefinition:
    Type: AWS::ECS::TaskDefinition
    Properties:
      Family: my-task
      ContainerDefinitions:
        - Name: my-container
          Image: my-docker-image
          Memory: 512
```

```
        Cpu: 256
        Essential: true

  MyService:
    Type: AWS::ECS::Service
    Properties:
      Cluster: !Ref MyCluster
      DesiredCount: 1
      TaskDefinition: !Ref MyTaskDefinition
      LaunchType: FARGATE
      NetworkConfiguration:
        AwsvpcConfiguration:
          Subnets:
            - subnet-id
          SecurityGroups:
            - sg-id
          AssignPublicIp: ENABLED
```

Use this template with the AWS CloudFormation console or CLI to provision ECS resources:
aws cloudformation create-stack --stack-name my-stack --template-body file://ecs-template.yaml

---

4. Terraform Configuration

Here's a Terraform script to create an ECS cluster, task definition, and service.

hcl

```
provider "aws" {
  region = "us-east-1"
}
```

```
resource "aws_ecs_cluster" "example" {
  name = "my-cluster"
}

resource "aws_ecs_task_definition" "example" {
  family                   = "my-task"
  container_definitions    = <<DEFINITION
[
  {
    "name": "my-container",
    "image": "my-docker-image",
    "memory": 512,
    "cpu": 256,
    "essential": true
  }
]
DEFINITION
}

resource "aws_ecs_service" "example" {
  name            = "my-service"
  cluster         = aws_ecs_cluster.example.id
  task_definition = aws_ecs_task_definition.example.arn
  desired_count   = 1
  launch_type     = "FARGATE"

  network_configuration {
    subnets         = ["subnet-id"]
    security_groups = ["sg-id"]
    assign_public_ip = true
  }
}
```

To apply the Terraform configuration:

Initialize Terraform:
terraform init
Apply the configuration:
terraform apply

---

5. Configuration with Pulumi

To configure Amazon ECS (Elastic Container Service) with **Pulumi**, you can use the Pulumi AWS SDK to define and manage your ECS cluster, task definitions, and services. Pulumi enables you to use programming languages like JavaScript, TypeScript, Python, Go, and C# for infrastructure as code. Here's an example using Python:

1. Install Pulumi and AWS SDK

First, install Pulumi and the AWS SDK:

pip install pulumi
pip install pulumi_aws

2. Configure ECS with Pulumi (Python Example)

Here’s a Python example to configure ECS using Pulumi.

a) Create ECS Cluster
python
import pulumi
import pulumi_aws as aws

 # Create an ECS Cluster
ecs_cluster = aws.ecs.Cluster("my-cluster")

b) Create a Task Definition

python

```
# Create an ECS Task Definition
task_definition = aws.ecs.TaskDefinition("my-task",
    family="my-task",
    cpu="256",  # 0.25 vCPU
    memory="512",  # 512 MiB of memory
    network_mode="awsvpc",
    requires_compatibilities=["FARGATE"],
    container_definitions=pulumi.Output.all().apply(lambda args: [
        {
            "name": "my-container",
            "image": "my-docker-image",
            "memory": 512,
            "cpu": 256,
            "essential": True
        }
    ])
)
```

c) Create ECS Service

python

```
# Create ECS Service
service = aws.ecs.Service("my-service",
    cluster=ecs_cluster.id,
    task_definition=task_definition.arn,
    desired_count=1,
    launch_type="FARGATE",
    network_configuration={
        "awsvpc_configuration": {
            "subnets": ["subnet-id"],  # Replace with your subnet ID
```

```
            "security_groups": ["sg-id"],  # Replace with your security group ID
            "assign_public_ip": "ENABLED"
        }
    }
)
```

d) Export Outputs

You can export the ECS cluster name, task definition, and service name as outputs:

python

```
pulumi.export("ecs_cluster_name", ecs_cluster.name)
pulumi.export("task_definition_name", task_definition.family)
pulumi.export("service_name", service.name)
```

3. Deploy with Pulumi

After defining the infrastructure in your Pulumi stack, you can deploy it using the following commands:

Initialize Pulumi (if not already initialized):
pulumi stack init my-stack

Deploy the ECS resources:
pulumi up

---

4. Pulumi with TypeScript Example

If you prefer TypeScript, here's a similar example:

typescript

```
import * as aws from "@pulumi/aws";

// Create an ECS Cluster
const ecsCluster = new aws.ecs.Cluster("my-cluster");

// Create ECS Task Definition
const taskDefinition = new aws.ecs.TaskDefinition("my-task", {
    family: "my-task",
    cpu: "256",  // 0.25 vCPU
    memory: "512",  // 512 MiB of memory
    networkMode: "awsvpc",
    requiresCompatibilities: ["FARGATE"],
    containerDefinitions: JSON.stringify([{
        name: "my-container",
        image: "my-docker-image",
        memory: 512,
        cpu: 256,
        essential: true
    }]),
});

// Create ECS Service
const ecsService = new aws.ecs.Service("my-service", {
    cluster: ecsCluster.id,
    taskDefinition: taskDefinition.arn,
    desiredCount: 1,
    launchType: "FARGATE",
    networkConfiguration: {
        awsvpcConfiguration: {
            subnets: ["subnet-id"],  // Replace with your subnet ID
            securityGroups: ["sg-id"],  // Replace with your security group ID
            assignPublicIp: "ENABLED"
        }
    }
```

```
});

export const ecsClusterName = ecsCluster.name;
export const taskDefinitionName = taskDefinition.family;
export const serviceName = ecsService.name;
```

To deploy this TypeScript code:

Install Pulumi dependencies:
npm install @pulumi/pulumi @pulumi/aws

Deploy:
pulumi up

---

4. EKS (Elastic Kubernetes Service): Managed Kubernetes service for containerized applications.

Introduction to EKS (Elastic Kubernetes Service)

EKS is a managed service by AWS that simplifies the process of running Kubernetes clusters on AWS without needing to install and operate your own Kubernetes control plane. It automatically handles tasks such as patching, scaling, and managing the Kubernetes master nodes. It is integrated with AWS services like IAM, VPC, CloudWatch, and more, providing a secure and scalable environment for running containerized applications.

---

1. AWS UI (Console)

- **Step 1**: Go to the EKS Console.

- **Step 2**: Click **Create Cluster**.
- **Step 3**: Provide a **Cluster Name**, **Kubernetes Version**, and choose a **VPC** and **Subnets**.
- **Step 4**: Configure the **IAM role** for the EKS cluster.
- **Step 5**: Select **Create Cluster**. The EKS cluster will start provisioning.

---

2. AWS CLI

- **Step 1**: Install and configure AWS CLI.

Step 2: Use the following command to create a cluster:

aws eks create-cluster --name my-cluster --role-arn arn:aws:iam::<account-id>:role/EKSClusterRole --resources-vpc-config subnetIds=subnet-xxxxxx,securityGroupIds=sg-xxxxxx

Step 3: Once the cluster is created, you can update your kubeconfig:
aws eks update-kubeconfig --name my-cluster

---

3. AWS CloudFormation Configuration

CloudFormation allows you to define and provision the infrastructure needed for EKS using templates.

Step 1: Create a CloudFormation template for EKS, such as:
yaml

```
Resources:
  EKSCluster:
    Type: AWS::EKS::Cluster
    Properties:
      Name: my-cluster
      RoleArn: arn:aws:iam::<account-id>:role/EKSClusterRole
      ResourcesVpcConfig:
```

```
      SubnetIds:
        - subnet-xxxxxx
      SecurityGroupIds:
        - sg-xxxxxx
```

Step 2: Create the stack using AWS CLI:
aws cloudformation create-stack --stack-name my-eks-cluster --template-body file://eks-cluster-template.yaml

---

4. Configuration with Terraform

Terraform provides an easy way to manage infrastructure as code.

Step 1: Create a main.tf configuration for EKS:
hcl

```
provider "aws" {
  region = "us-west-2"
}

resource "aws_eks_cluster" "example" {
  name     = "my-cluster"
  role_arn = "arn:aws:iam::<account-id>:role/EKSClusterRole"
  vpc_config {
    subnet_ids = ["subnet-xxxxxx"]
    security_group_ids = ["sg-xxxxxx"]
  }
}
```

Step 2: Initialize Terraform and apply the configuration:
terraform init
terraform apply

---

5. Pulumi Configuration

Pulumi allows you to define cloud infrastructure using modern programming languages.

- **Step 1**: Install Pulumi and set up the project.

Step 2: Define the EKS cluster in your Pulumi code:
typescript

```
import * as aws from "@pulumi/aws";
import * as eks from "@pulumi/eks";

const cluster = new eks.Cluster("my-cluster", {
   instanceType: "t3.medium",
   desiredCapacity: 2,
   minSize: 1,
   maxSize: 3,
});
```

- **Step 3**: Run pulumi up to provision the cluster.

---

5. Fargate: Serverless compute engine for containers.

1. AWS UI Configuration

To configure AWS Fargate via the AWS Management Console:

- **Step 1**: Go to the **Amazon ECS** console.
- **Step 2**: Create a new **Cluster** or choose an existing one.
- **Step 3**: Select **Create Service**, choose **Fargate** as the launch type, and configure your task definition.
- **Step 4**: Set task parameters like CPU, memory, and networking options.
- **Step 5**: Define your container's image, ports, and environment variables.

---

2. AWS CLI Configuration

To configure AWS Fargate using the AWS CLI, use the following commands:

Step 1: Register a task definition:

```
aws ecs register-task-definition \
   --family my-fargate-task \
   --execution-role-arn arn:aws:iam::aws_account_id:role/my-fargate-role \
   --network-mode awsvpc \
   --container-definitions '[{
      "name": "my-container",
      "image": "my-image:latest",
      "memory": 512,
      "cpu": 256,
      "essential": true
   }]'
```

Step 2: Create a service:

```
aws ecs create-service \
   --cluster my-cluster \
   --service-name my-fargate-service \
   --task-definition my-fargate-task \
   --desired-count 1 \
   --launch-type FARGATE \
```

```
  --network-configuration
"awsvpcConfiguration={subnets=[subnet-xxxxxxx],securityGroups=[sg-xxxxxxx],
assignPublicIp=ENABLED}"
```

---

3. CloudFormation Configuration

For AWS Fargate with CloudFormation, create a template to define your ECS service:

```
yaml
Resources:
  MyFargateCluster:
    Type: AWS::ECS::Cluster
    Properties:
      ClusterName: MyFargateCluster

  MyFargateService:
    Type: AWS::ECS::Service
    Properties:
      Cluster: !Ref MyFargateCluster
      DesiredCount: 1
      LaunchType: FARGATE
      TaskDefinition: !Ref MyFargateTaskDefinition
      NetworkConfiguration:
        AwsvpcConfiguration:
          Subnets:
            - subnet-xxxxxxx
          SecurityGroups:
            - sg-xxxxxxx
          AssignPublicIp: ENABLED

  MyFargateTaskDefinition:
```

```
  Type: AWS::ECS::TaskDefinition
  Properties:
    Family: my-fargate-task
    ExecutionRoleArn: arn:aws:iam::aws_account_id:role/my-fargate-role
    ContainerDefinitions:
      - Name: my-container
        Image: my-image:latest
        Memory: 512
        Cpu: 256
```

---

4. Terraform Configuration

For Terraform, the configuration for AWS Fargate would look like this:

hcl

```
resource "aws_ecs_cluster" "my_fargate_cluster" {
  name = "my-fargate-cluster"
}

resource "aws_ecs_task_definition" "my_fargate_task" {
  family                = "my-fargate-task"
  execution_role_arn    = "arn:aws:iam::aws_account_id:role/my-fargate-role"
  network_mode          = "awsvpc"

  container_definitions = jsonencode([{
    name      = "my-container"
    image     = "my-image:latest"
    memory    = 512
    cpu       = 256
    essential = true
  }])
}
```

```
resource "aws_ecs_service" "my_fargate_service" {
  name            = "my-fargate-service"
  cluster         = aws_ecs_cluster.my_fargate_cluster.id
  task_definition = aws_ecs_task_definition.my_fargate_task.arn
  desired_count   = 1
  launch_type     = "FARGATE"
  network_configuration {
    awsvpc_configuration {
      subnets          = ["subnet-xxxxxxx"]
      security_groups  = ["sg-xxxxxxx"]
      assign_public_ip = "ENABLED"
    }
  }
}
```

---

5. Pulumi Configuration

In Pulumi, the configuration to deploy AWS Fargate looks like this (using Python):

python

```
import pulumi
from pulumi_aws import ecs, iam

# Create an ECS Cluster
cluster = ecs.Cluster("my-fargate-cluster")

# Create a Task Definition
task_definition = ecs.TaskDefinition("my-fargate-task",
    family="my-fargate-task",
    execution_role_arn="arn:aws:iam::aws_account_id:role/my-fargate-role",
```

```
    network_mode="awsvpc",
    container_definitions='[{"name": "my-container", "image": "my-image:latest",
"memory": 512, "cpu": 256, "essential": true}]'
)

# Create ECS Service
service = ecs.Service("my-fargate-service",
    cluster=cluster.id,
    task_definition=task_definition.arn,
    desired_count=1,
    launch_type="FARGATE",
    network_configuration=ecs.ServiceNetworkConfigurationArgs(

awsvpc_configuration=ecs.ServiceNetworkConfigurationAwsvpcConfigurationArg
s(
            subnets=["subnet-xxxxxxx"],
            security_groups=["sg-xxxxxxx"],
            assign_public_ip="ENABLED"
        )
    )
)
```

---

6. Batch: Fully managed batch processing at scale.

AWS Fully Managed Batch Processing

AWS Batch allows you to efficiently run hundreds to thousands of batch computing jobs on AWS. It dynamically provisions the optimal quantity and type of compute resources (e.g., EC2 instances or AWS Fargate) based on the volume and resource requirements of the batch jobs submitted.

---

1. AWS UI Configuration

1. **Navigate to AWS Batch Console**:
    - Go to the AWS Management Console, and search for **AWS Batch**.
2. **Create a Compute Environment**:
    - In the AWS Batch console, choose **Create compute environment**.
    - Define instance types, EC2 resources, and choose between Managed or Unmanaged compute environments.
3. **Create a Job Queue**:
    - In the **Job queues** section, click **Create job queue** and associate it with the compute environment.
4. **Create a Job Definition**:
    - Under **Job definitions**, click **Create job definition** to specify parameters like job role, Docker image, and resource requirements.
5. **Submit Jobs**:
    - Submit jobs from the **Job queues** section by selecting the job definition and providing necessary parameters.

---

2. AWS CLI Configuration

You can manage AWS Batch jobs using the **AWS CLI**.

Create Compute Environment:
aws batch create-compute-environment --compute-environment-name MyComputeEnv --type MANAGED --compute-resources type=EC2,instanceRole=ecsInstanceRole,instanceTypes=m4.large,maxvCpus=16

Create Job Queue:
aws batch create-job-queue --job-queue-name MyJobQueue --priority 1 --compute-environments computeEnvironment=MyComputeEnv

Submit Job:
aws batch submit-job --job-name MyBatchJob --job-queue MyJobQueue
--job-definition MyJobDefinition

---

3. CloudFormation Configuration

CloudFormation allows you to define the AWS Batch environment as code. Below is an example configuration:

yaml

```
Resources:
  MyComputeEnvironment:
    Type: AWS::Batch::ComputeEnvironment
    Properties:
      ComputeEnvironmentName: MyComputeEnv
      Type: MANAGED
      ComputeResources:
        Type: EC2
        InstanceRole: ecsInstanceRole
        InstanceTypes:
          - m4.large
        MaxvCpus: 16
      State: ENABLED

  MyJobQueue:
    Type: AWS::Batch::JobQueue
    Properties:
      JobQueueName: MyJobQueue
      Priority: 1
      ComputeEnvironmentOrder:
        - Order: 1
```

```
      ComputeEnvironment: !Ref MyComputeEnvironment

  MyJobDefinition:
    Type: AWS::Batch::JobDefinition
    Properties:
      JobDefinitionName: MyJobDefinition
      Type: container
      ContainerProperties:
        Image: "my-docker-image"
        Vcpus: 2
        Memory: 4096
```

---

4. Terraform Configuration

To configure AWS Batch using **Terraform**, define resources like aws_batch_compute_environment, aws_batch_job_queue, and aws_batch_job_definition in a Terraform script.

hcl

```
resource "aws_batch_compute_environment" "my_compute_env" {
  compute_environment_name = "MyComputeEnv"
  type                     = "MANAGED"

  compute_resources {
    type           = "EC2"
    instance_role  = "ecsInstanceRole"
    instance_types = ["m4.large"]
    max_vcpus      = 16
  }
}

resource "aws_batch_job_queue" "my_job_queue" {
```

```
  name                = "MyJobQueue"
  priority            = 1
  compute_environment_order {
    order             = 1
    compute_environment =
aws_batch_compute_environment.my_compute_env.arn
  }
}

resource "aws_batch_job_definition" "my_job_definition" {
  name = "MyJobDefinition"
  type = "container"

  container_properties {
    image = "my-docker-image"
    vcpus = 2
    memory = 4096
  }
}
```

---

5. Pulumi Configuration

For **Pulumi**, you can use the following code to create an AWS Batch environment:

javascript

```
const aws = require("@pulumi/aws");

const computeEnvironment = new
aws.batch.ComputeEnvironment("myComputeEnv", {
    type: "MANAGED",
    computeResources: {
        type: "EC2",
```

```
        instanceRole: "ecsInstanceRole",
        instanceTypes: ["m4.large"],
        maxVcpus: 16,
    },
});

const jobQueue = new aws.batch.JobQueue("myJobQueue", {
    priority: 1,
    computeEnvironmentOrder: [{
        order: 1,
        computeEnvironment: computeEnvironment.arn,
    }],
});

const jobDefinition = new aws.batch.JobDefinition("myJobDefinition", {
    type: "container",
    containerProperties: {
        image: "my-docker-image",
        vcpus: 2,
        memory: 4096,
    },
});
```

---

7. Elastic Beanstalk: Platform-as-a-Service (PaaS) for deploying and managing applications.

Elastic Beanstalk Overview

AWS Elastic Beanstalk is a Platform-as-a-Service (PaaS) that makes it easy to deploy and manage applications in the cloud without worrying about the infrastructure. It supports several programming languages and frameworks, such as Java, .NET, PHP, Node.js, Python, Ruby, and more. Elastic Beanstalk automates

deployment, scaling, and monitoring, while providing an easy-to-use interface for managing your application lifecycle.

---

1. AWS UI Configuration

To configure Elastic Beanstalk via the AWS Management Console:

1. Go to the **Elastic Beanstalk** service in the AWS Management Console.
2. Click **Create Application**.
3. Choose the platform (e.g., Node.js, Python, Java) for your application.
4. Configure the environment (e.g., Web Server, Worker, etc.).
5. Upload your application code or specify the source (e.g., GitHub or S3).
6. Review and create the environment.

---

2. AWS CLI Configuration

To deploy your application using the AWS CLI:

Initialize Elastic Beanstalk:
eb init -p python-3.7 my-app

Create an environment:
eb create my-app-env

Deploy your application:
eb deploy

View the application URL:
eb open

---

3. CloudFormation Configuration

Here’s an example of a CloudFormation template to create an Elastic Beanstalk environment:

yaml

```
Resources:
  MyElasticBeanstalkApplication:
    Type: AWS::ElasticBeanstalk::Application
    Properties:
      ApplicationName: my-app

  MyElasticBeanstalkEnvironment:
    Type: AWS::ElasticBeanstalk::Environment
    Properties:
      ApplicationName: !Ref MyElasticBeanstalkApplication
      EnvironmentName: my-app-env
      SolutionStackName: "64bit Amazon Linux 2 v3.3.6 running Python 3.8"
      VersionLabel: v1
```

Deploy the CloudFormation template using the AWS CLI:

```
aws cloudformation create-stack --stack-name my-stack --template-body file://template.yaml
```

---

4. Terraform Configuration

Here's a Terraform example to configure Elastic Beanstalk:

hcl

```
provider "aws" {
  region = "us-west-2"
}

resource "aws_elastic_beanstalk_application" "example" {
  name = "my-app"
}

resource "aws_elastic_beanstalk_environment" "example" {
  name                = "my-app-env"
  application         = aws_elastic_beanstalk_application.example.name
  solution_stack_name = "64bit Amazon Linux 2 v3.3.6 running Python 3.8"
  version_label       = "v1"
}
```

Run Terraform to deploy:

```
terraform init
terraform apply
```

---

5. Pulumi Configuration

With Pulumi, you can create Elastic Beanstalk resources using the AWS SDK:

javascript

```
const aws = require("@pulumi/aws");

const app = new aws.elasticbeanstalk.Application("my-app", {
    name: "my-app",
});
```

```
const env = new aws.elasticbeanstalk.Environment("my-app-env", {
    name: "my-app-env",
    application: app.name,
    solutionStackName: "64bit Amazon Linux 2 v3.3.6 running Python 3.8",
    versionLabel: "v1",
});
```

To deploy using Pulumi:

```
pulumi up
```

---

8. Outposts: AWS infrastructure extension to on-premises locations.

Introduction:

AWS Outposts is a fully managed service that extends AWS infrastructure, services, APIs, and tools to your on-premises environment. This enables you to run hybrid workloads seamlessly, using the same AWS APIs, tools, and infrastructure across both on-premises and AWS cloud environments. You can configure AWS Outposts in various ways using the AWS UI, CLI, CloudFormation, Terraform, and Pulumi.

---

1. AWS UI Configuration:

1. **Navigate to the AWS Management Console** and search for **Outposts**.
2. Click on **Create Outpost**.
3. Select the **Region**, **Availability Zone**, and specify the **Outpost location**.
4. Configure the **capacity** required (e.g., EC2 instances) and any additional settings such as **network connectivity**.

5. Follow the steps to review and confirm the creation of your Outpost.

---

2. AWS CLI Configuration:

You can use the AWS CLI to manage AWS Outposts resources. First, ensure you have the AWS CLI installed and configured.

Create Outpost:

aws outposts create-outpost --name "MyOutpost" --site-id "site-id" --availability-zone "us-west-2a" --outpost-type "rack"

List Outposts:

aws outposts list-outposts

Describe Outpost:

aws outposts describe-outpost --outpost-id "outpost-id"

---

3. AWS CloudFormation Configuration:

In AWS CloudFormation, you can manage your Outposts as part of a stack by defining it in a template.

yaml

```
Resources:
  MyOutpost:
    Type: 'AWS::Outposts::Outpost'
    Properties:
      Name: "MyOutpost"
```

```
SiteId: "site-id"
AvailabilityZone: "us-west-2a"
OutpostType: "rack"
```

Run the CloudFormation template to create the Outpost:

aws cloudformation create-stack --stack-name MyOutpostStack --template-body file://outpost-template.yml

---

4. Terraform Configuration:

Terraform can be used to create and manage Outposts resources.

Example Terraform Configuration:

```
hcl
provider "aws" {
  region = "us-west-2"
}

resource "aws_outposts_outpost" "example" {
  name            = "MyOutpost"
  site_id         = "site-id"
  availability_zone = "us-west-2a"
  outpost_type    = "rack"
}
```

To create the Outpost:

```
terraform init
terraform apply
```

---

5. Pulumi Configuration:

Pulumi allows you to manage AWS Outposts with your preferred programming language.

Example Pulumi (TypeScript) Configuration:

typescript

```
import * as aws from "@pulumi/aws";

const outpost = new aws.outposts.Outpost("MyOutpost", {
    siteId: "site-id",
    availabilityZone: "us-west-2a",
    outpostType: "rack",
});
```

Deploy the Outpost with Pulumi:

pulumi up

---

9. Lightsail: Simplified cloud computing for small businesses and developers.

AWS Lightsail: Simplified Cloud Computing for Small Businesses and Developers

AWS Lightsail is a cloud computing service that offers an easy-to-use interface for developers and small businesses to manage virtual private servers (VPS), databases, networking, and other resources. It simplifies cloud management by providing a straightforward pricing model and pre-configured solutions, making it ideal for small-scale applications and websites.

---

1. Configuration with AWS UI

To configure Lightsail through the AWS Management Console:

1. Sign in to the AWS Management Console.
2. Navigate to **Lightsail** from the services menu.
3. Click **Create instance** to start the configuration.
4. Choose an operating system or application blueprint (e.g., WordPress, LAMP, etc.).
5. Select the instance plan that fits your requirements (e.g., cost-effective options for small projects).
6. Set a name for your instance, configure networking options, and select the key pair for SSH access.
7. Click **Create** to launch your Lightsail instance.

---

2. Configuration with AWS CLI

To create a Lightsail instance using the AWS CLI:

```
aws lightsail create-instances \
   --instance-names MyLightsailInstance \
   --availability-zone us-east-1a \
   --blueprint-id amazon_linux_2 \
   --bundle-id micro_1_0 \
   --key-pair-name MyKeyPair
```

This command will create a Lightsail instance with the specified parameters (e.g., availability zone, blueprint, bundle, etc.).

---

3. Configuration with CloudFormation

A CloudFormation template to deploy a Lightsail instance might look like this:

yaml
```
Resources:
  MyLightsailInstance:
    Type: 'AWS::Lightsail::Instance'
    Properties:
      InstanceName: 'MyLightsailInstance'
      AvailabilityZone: 'us-east-1a'
      BlueprintId: 'amazon_linux_2'
      BundleId: 'micro_1_0'
      KeyPairName: 'MyKeyPair'
```

You can deploy this template via the AWS CloudFormation console or CLI.

---

4. Configuration with Terraform

To configure Lightsail using Terraform:

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_lightsail_instance" "my_instance" {
  instance_name = "MyLightsailInstance"
  availability_zone = "us-east-1a"
  blueprint_id = "amazon_linux_2"
  bundle_id = "micro_1_0"
  key_pair_name = "MyKeyPair"
}
```

You can then apply this configuration using:

terraform init
terraform apply

---

5. Configuration with Pulumi

To configure Lightsail using Pulumi (in JavaScript/TypeScript):

javascript

```
const aws = require("@pulumi/aws");

const lightsailInstance = new aws.lightsail.Instance("myInstance", {
    availabilityZone: "us-east-1a",
    blueprintId: "amazon_linux_2",
    bundleId: "micro_1_0",
    instanceName: "MyLightsailInstance",
    keyPairName: "MyKeyPair",
});
```

You can then deploy the infrastructure with Pulumi:

pulumi up

Storage

1. S3 (Simple Storage Service): Object storage.

AWS S3 (Simple Storage Service) – Object Storage

AWS S3 provides scalable object storage for a wide variety of use cases such as backup, data archiving, content distribution, and more. It offers high durability and availability for storing and retrieving data.

---

1. AWS Console (UI) Configuration

1. Log into the **AWS Management Console**.
2. Navigate to **S3** under the "Storage" section.
3. Click **Create bucket**.
4. Set the **Bucket name** (unique globally) and select a region.
5. Configure other settings (versioning, encryption, access permissions).
6. Click **Create** to complete the bucket setup.

---

2. AWS CLI Configuration

To create a bucket using AWS CLI:

aws s3 mb s3://my-unique-bucket-name --region us-east-1

- Replace my-unique-bucket-name with your desired bucket name.
- Ensure AWS CLI is installed and configured with appropriate credentials.

---

3. CloudFormation Configuration

To create an S3 bucket using AWS CloudFormation, define the following in a CloudFormation template (s3.yaml):

yaml

```
Resources:
  MyS3Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: my-unique-bucket-name
```

Deploy using the AWS CLI:

aws cloudformation create-stack --stack-name my-s3-stack --template-body file://s3.yaml

---

4. Terraform Configuration

Terraform configuration to create an S3 bucket:

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_s3_bucket" "my_s3_bucket" {
  bucket = "my-unique-bucket-name"
  acl    = "private"
```

```
}
```

Run the following Terraform commands:

```
terraform init
terraform apply
```

---

5. Pulumi Configuration

Pulumi allows you to define infrastructure using code. Here's how to configure an S3 bucket with Pulumi in JavaScript:

javascript

```
const aws = require("@pulumi/aws");

const bucket = new aws.s3.Bucket("myS3Bucket", {
    bucket: "my-unique-bucket-name"
});
```

To deploy:

```
pulumi up
```

---

2. EBS (Elastic Block Store): Block storage for EC2.

Introduction:

AWS Elastic Block Store (EBS) is a scalable, high-performance block storage service designed to be used with Amazon EC2 instances. It provides persistent storage that can be attached to EC2 instances, allowing data to persist even if the instance is terminated. EBS volumes are ideal for use cases such as databases, file systems, and other enterprise applications that require reliable and fast storage.

---

1. **AWS Console (UI) Configuration:**

1. Go to the AWS Management Console.
2. Navigate to **EC2** and select **Volumes** under **Elastic Block Store**.
3. Click on **Create Volume**.
4. Choose the volume type (e.g., General Purpose SSD) and configure the size, availability zone, and other parameters.
5. After the volume is created, select the volume and click on **Actions** > **Attach Volume**.
6. Choose the EC2 instance to which the volume will be attached and click **Attach**.

---

2. AWS CLI Configuration:

Create an EBS Volume:
aws ec2 create-volume --size 10 --volume-type gp2 --availability-zone us-east-1a

Attach the EBS Volume to an EC2 Instance:
aws ec2 attach-volume --volume-id vol-xxxxxxxx --instance-id i-xxxxxxxx --device /dev/sdf

---

3. AWS CloudFormation Configuration:

yaml

Resources:

```
  MyEBSVolume:
    Type: AWS::EC2::Volume
    Properties:
      AvailabilityZone: us-east-1a
      Size: 10
      VolumeType: gp2

  VolumeAttachment:
    Type: AWS::EC2::VolumeAttachment
    Properties:
      InstanceId: i-xxxxxxxx
      VolumeId: !Ref MyEBSVolume
      Device: /dev/sdf
```

---

4. Terraform Configuration:

hcl

```
resource "aws_ebs_volume" "example" {
  availability_zone = "us-east-1a"
  size              = 10
  type              = "gp2"
}

resource "aws_volume_attachment" "example_attachment" {
  device_name = "/dev/sdf"
  volume_id   = aws_ebs_volume.example.id
  instance_id = "i-xxxxxxxx"
}
```

---

5. Pulumi Configuration:

typescript

```
import * as aws from "@pulumi/aws";

// Create an EBS volume
const volume = new aws.ebs.Volume("myVolume", {
    availabilityZone: "us-east-1a",
    size: 10,
    type: "gp2",
});

// Attach the volume to an EC2 instance
const attachment = new aws.ec2.VolumeAttachment("myVolumeAttachment", {
    volumeId: volume.id,
    instanceId: "i-xxxxxxxx",
    device: "/dev/sdf",
});
```

---

3. EFS (Elastic File System): Managed file storage.

Introduction to EFS:

Amazon Elastic File System (EFS) is a scalable and fully managed Network File System (NFS) designed to be used with Amazon Web Services (AWS) cloud services and on-premises resources. It provides a simple, scalable, elastic file system for use with AWS Cloud services and on-premises resources. EFS can scale up or down automatically as you add or remove files, making it a great choice for applications that require a highly available and scalable file storage system.

EFS Configuration:

1. **AWS Management Console (UI):**
    - Open the **Amazon EFS console**.

- Click on **Create file system**.
- Choose the **VPC** where you want to create the EFS.
- Select the **Availability and Performance Mode** (General Purpose or Max I/O).
- Configure the **Access Points** (optional, for shared access).
- Choose the **Network Access** (Security Group and Mount Targets).
- Review the configuration and click **Create File System**.

---

2. AWS CLI: You can create an EFS using the AWS CLI with the following command:

```
aws efs create-file-system --performance-mode generalPurpose --tags Key=Name,Value=MyEFS
```

For mounting EFS:

```
sudo mount -t nfs4 -o nfsvers=4.1 <EFS-DNS-NAME>:/ /mnt/efs
```

---

3. CloudFormation: Use AWS CloudFormation to create an EFS file system. Here's an example of a CloudFormation template:

```
yaml
Resources:
  MyEFSFileSystem:
    Type: 'AWS::EFS::FileSystem'
    Properties:
      PerformanceMode: generalPurpose
      Encrypted: true
```

2. This will create an EFS with encryption enabled and the General Purpose performance mode.

4.Terraform:

The Terraform script below creates an EFS resource:
hcl

```
provider "aws" {
  region = "us-west-2"
}

resource "aws_efs_file_system" "example" {
  performance_mode = "generalPurpose"
  encrypted        = true
}

resource "aws_security_group" "example" {
  name        = "efs_security_group"
  description = "Security group for EFS"
}
```

---

5. Pulumi (TypeScript example): Pulumi can be used to create an EFS using the following script:
typescript

```
import * as aws from "@pulumi/aws";

const fileSystem = new aws.efs.FileSystem("myEFS", {
  performanceMode: "GeneralPurpose",
  encrypted: true,
});
```

export const fileSystemId = fileSystem.id;

---

4. FSx: Managed file systems (Windows, Lustre, NetApp).

AWS FSx: Managed File Systems (Windows, Lustre, NetApp)

AWS FSx provides fully managed, high-performance file systems to support workloads across various use cases. It offers options like **Windows File Server**, **Lustre**, and **NetApp ONTAP** to meet diverse storage requirements.

---

1. AWS UI Configuration

To create a new FSx file system using the AWS Management Console:

1. Go to the **FSx** dashboard.
2. Click **Create file system**.
3. Choose the file system type (e.g., **Windows File Server**, **Lustre**, **NetApp ONTAP**).
4. Provide configuration details such as storage capacity, VPC settings, and backup options.
5. Follow the prompts to create the file system.

---

2. AWS CLI Configuration

Using AWS CLI, you can create an FSx file system. Example for a **Windows File Server**:

```
aws fsx create-file-system \
    --file-system-type WINDOWS \
    --subnet-id subnet-xxxxxxxx \
    --storage-capacity 300 \
    --windows-configuration
ActiveDirectoryId=d-xxxxxxx,SelfManagedADConfiguration={UserName=fsx-us
er,Password=Password123}
```

---

3. CloudFormation Configuration

In CloudFormation, you can define an FSx file system in your YAML template:

yaml

```
Resources:
  MyFSxFileSystem:
    Type: AWS::FSx::FileSystem
    Properties:
      FileSystemType: WINDOWS
      StorageCapacity: 300
      SubnetIds:
        - subnet-xxxxxxxx
      WindowsConfiguration:
        ActiveDirectoryId: d-xxxxxxx
        SelfManagedADConfiguration:
          UserName: fsx-user
          Password: Password123
```

---

4. Terraform Configuration

Using Terraform, you can define an FSx file system as follows:

hcl

```
resource "aws_fsx_windows_file_system" "example" {
  subnet_id          = "subnet-xxxxxxxx"
  storage_capacity   = 300
  windows_configuration {
    active_directory_id = "d-xxxxxxx"
    self_managed_active_directory_configuration {
      user_name = "fsx-user"
      password  = "Password123"
    }
  }
}
```

---

5. Pulumi Configuration (TypeScript)

For Pulumi, you can configure FSx like this:

typescript

```
import * as aws from "@pulumi/aws";

const fsxFileSystem = new aws.fsx.WindowsFileSystem("fsxFileSystem", {
    subnetId: "subnet-xxxxxxxx",
    storageCapacity: 300,
    windowsConfiguration: {
        activeDirectoryId: "d-xxxxxxx",
        selfManagedActiveDirectoryConfiguration: {
            userName: "fsx-user",
            password: "Password123",
        },
    },
```

});

---

5. Glacier: Low-cost archival storage.

Amazon Glacier is a low-cost cloud storage service for data archiving and long-term backup. It is designed to provide secure, durable, and extremely low-cost storage for data that is infrequently accessed. The service is ideal for storing large amounts of data, such as backups and archives, that don't require real-time access.

---

1. Configuration via AWS UI:

1. **Login to AWS Console:** Go to the AWS Management Console and sign in.
2. **Navigate to Glacier:** In the services search bar, type **S3 Glacier** and select it.
3. **Create Vault:**
    - Click on **Create vault**.
    - Provide a name for the vault and select the **region**.
    - Optionally, you can configure vault lock policies.
4. **Upload Archives:**
    - Once the vault is created, you can upload archives to Glacier through the **Amazon S3 Glacier console**.

---

2. Configuration via AWS CLI:

Create Vault:

aws glacier create-vault --vault-name my-vault --region us-west-2

Upload Archive:
aws glacier upload-archive --vault-name my-vault --body /path/to/your/file --archive-description "Backup Archive"

---

3. Configuration via CloudFormation:

You can use the following CloudFormation template to create a Glacier vault.

yaml

```
Resources:
  MyGlacierVault:
    Type: "AWS::Glacier::Vault"
    Properties:
      VaultName: "my-vault"
```

- Deploy this CloudFormation template using AWS CLI or AWS Management Console to create the Glacier vault.

---

4. Configuration via Terraform:

To create an Amazon Glacier vault using Terraform, you can define a resource block like this:

hcl

```
provider "aws" {
  region = "us-west-2"
}
```

```
resource "aws_glacier_vault" "my_vault" {
  name = "my-vault"
}
```

Initialize Terraform:
terraform init

Apply Configuration:
terraform apply

---

5. Configuration via Pulumi:

For Pulumi, you can define a Glacier vault like this:

typescript

```
import * as aws from "@pulumi/aws";

const myVault = new aws.glacier.Vault("my-vault", {
    vaultName: "my-vault",
});
```

Install Pulumi AWS package:
npm install @pulumi/aws

- **Run Pulumi:**
  pulumi up

---

6. Storage Gateway: Hybrid cloud storage.

Introduction to AWS Storage Gateway: Hybrid Cloud Storage

AWS Storage Gateway is a hybrid cloud storage service that enables on-premises applications to seamlessly integrate with cloud storage. It allows enterprises to connect their on-premises IT infrastructure with cloud storage for backup, archiving, and disaster recovery.

AWS Storage Gateway provides three key configurations:

1. **File Gateway**: For storing files in S3.
2. **Tape Gateway**: For integrating with virtual tapes in S3.
3. **Volume Gateway**: For integrating block storage volumes with cloud storage.

---

1. AWS UI Configuration

- Sign in to the AWS Management Console.
- Navigate to **Storage Gateway** under the **Storage** category.
- Choose **Create gateway**, select the type (File, Tape, or Volume Gateway), and follow the setup wizard to configure the gateway.
- For **File Gateway**, select the local storage, and for **Volume or Tape Gateway**, follow the prompts to map your on-premises storage to cloud volumes or tapes.

---

2. AWS CLI Configuration

To configure a **File Gateway** with the AWS CLI, you can use the following commands:

aws storagegateway create-gateway --gateway-type FILE_S3 --gateway-name "MyFileGateway" --gateway-region us-west-2 --tape-drive-type "IBM-TS3500"

For a **Volume Gateway**:

aws storagegateway create-gateway --gateway-type VOLUME --gateway-name "MyVolumeGateway" --tape-drive-type "IBM-TS3500" --region us-west-2

---

3. CloudFormation Configuration

To create a **File Gateway** with CloudFormation, you would use a configuration like:

yaml

```
Resources:
  MyStorageGateway:
    Type: 'AWS::StorageGateway::Gateway'
    Properties:
      GatewayName: "MyFileGateway"
      GatewayType: "FILE_S3"
      GatewayRegion: "us-west-2"
```

Save this YAML file and create the stack using:

aws cloudformation create-stack --stack-name "StorageGatewayStack" --template-body file://gateway.yaml

---

4. Terraform Configuration

For Terraform, you can define a Storage Gateway resource in your .tf file:

hcl

```
resource "aws_storagegateway_gateway" "example" {
  gateway_name     = "MyFileGateway"
  gateway_type     = "FILE_S3"
  gateway_region   = "us-west-2"
  tape_drive_type  = "IBM-TS3500"
}
```

Apply the Terraform configuration:

```
terraform init
terraform apply
```

---

5. Pulumi Configuration

Using Pulumi in TypeScript, you can define the configuration as follows:

typescript

```
import * as aws from "@pulumi/aws";

const gateway = new aws.storagegateway.Gateway("MyFileGateway", {
    gatewayName: "MyFileGateway",
    gatewayType: "FILE_S3",
    gatewayRegion: "us-west-2",
    tapeDriveType: "IBM-TS3500"
});
```

Run the Pulumi stack:

```
pulumi up
```

Database

1. RDS (Relational Database Service): Managed relational databases.

Introduction to AWS RDS (Relational Database Service)

Amazon RDS (Relational Database Service) is a fully managed database service that simplifies database setup, operation, and scaling in the cloud. It supports multiple database engines, including MySQL, PostgreSQL, MariaDB, Oracle, and SQL Server. RDS automates tasks like backups, patching, scaling, and monitoring, allowing users to focus on application development.

---

1. AWS UI (Console) Configuration

1. Sign in to the AWS Management Console
2. Navigate to **RDS** → Click **Create Database**
3. Select **Standard Create**
4. Choose the **Database Engine** (e.g., MySQL, PostgreSQL)
5. Configure **DB instance settings** (DB identifier, username, password)
6. Choose **Instance Type, Storage, and VPC**
7. Enable or disable **Public Access**
8. Configure **Backup, Monitoring, and Maintenance** options
9. Click **Create Database**

---

2. AWS CLI Configuration

Create an RDS instance using AWS CLI:

sh
aws rds create-db-instance \

```
  --db-instance-identifier mydbinstance \
  --db-instance-class db.t3.micro \
  --engine mysql \
  --master-username admin \
  --master-user-password MySecurePassword \
  --allocated-storage 20
```

To list RDS instances:

```
aws rds describe-db-instances
```

To delete an RDS instance:

```
aws rds delete-db-instance --db-instance-identifier mydbinstance
--skip-final-snapshot
```

---

3. AWS CloudFormation Configuration

Create a rds-stack.yaml file:

yaml

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  MyDBInstance:
    Type: "AWS::RDS::DBInstance"
    Properties:
      DBInstanceIdentifier: "mydbinstance"
      DBInstanceClass: "db.t3.micro"
      Engine: "mysql"
      MasterUsername: "admin"
      MasterUserPassword: "MySecurePassword"
      AllocatedStorage: "20"
```

Deploy the stack:

aws cloudformation create-stack --stack-name MyRDSStack --template-body file://rds-stack.yaml

---

4. Terraform Configuration

Create a main.tf file:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_db_instance" "mydb" {
  identifier          = "mydbinstance"
  engine              = "mysql"
  instance_class      = "db.t3.micro"
  username            = "admin"
  password            = "MySecurePassword"
  allocated_storage   = 20
  skip_final_snapshot = true
}
```

Deploy the configuration:

terraform init
terraform apply -auto-approve

---

5. Pulumi Configuration (Python)

Install Pulumi and AWS SDK:

sh

```
pip install pulumi pulumi-aws
```

Create an __main__.py file:

python

```
import pulumi
import pulumi_aws as aws

db = aws.rds.Instance("mydbinstance",
    engine="mysql",
    instance_class="db.t3.micro",
    allocated_storage=20,
    username="admin",
    password="MySecurePassword",
    skip_final_snapshot=True
)

pulumi.export("db_endpoint", db.endpoint)
```

Deploy the configuration:

```
pulumi up
```

---

2. **DynamoDB**: NoSQL database.

Amazon DynamoDB: NoSQL Database

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. It is designed for applications requiring consistent, single-digit millisecond latency at any scale. DynamoDB supports key-value and document data models, making it ideal for use cases such as real-time applications, gaming, IoT, and serverless architectures.

DynamoDB Configuration Methods

1. AWS Management Console (UI)

- Go to **AWS Console → DynamoDB**
- Click **Create Table**
- Enter **Table Name** and **Primary Key (Partition Key & Optional Sort Key)**
- Configure **Read/Write capacity mode** (On-Demand or Provisioned)
- Enable **Point-in-time recovery**, **Encryption**, and **Auto Scaling** (if needed)
- Click **Create Table**

---

2. AWS CLI

Install and configure AWS CLI if not already installed:

aws configure

Create a DynamoDB table:

```
aws dynamodb create-table \
    --table-name MyTable \
    --attribute-definitions AttributeName=ID,AttributeType=S \
    --key-schema AttributeName=ID,KeyType=HASH \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

List all tables:

aws dynamodb list-tables

---

3. AWS CloudFormation (YAML)

Create a dynamodb-template.yaml file:

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  MyDynamoDBTable:
    Type: AWS::DynamoDB::Table
    Properties:
      TableName: MyTable
      AttributeDefinitions:
        - AttributeName: ID
          AttributeType: S
      KeySchema:
        - AttributeName: ID
          KeyType: HASH
      BillingMode: PAY_PER_REQUEST
```

Deploy the stack:

aws cloudformation create-stack --stack-name MyDynamoDBStack
--template-body file://dynamodb-template.yaml

---

4. Terraform Configuration

Create a dynamodb.tf file:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_dynamodb_table" "my_table" {
  name         = "MyTable"
  billing_mode = "PAY_PER_REQUEST"

  attribute {
    name = "ID"
    type = "S"
  }

  hash_key = "ID"
}
```

Deploy with:

```
terraform init
terraform apply -auto-approve
```

---

5. Pulumi (Python)

Install Pulumi and AWS SDK:

```
pip install pulumi pulumi-aws
```

Create a __main__.py file:

```
python

import pulumi
import pulumi_aws as aws
```

```
table = aws.dynamodb.Table("myTable",
    attributes=[{"name": "ID", "type": "S"}],
    hash_key="ID",
    billing_mode="PAY_PER_REQUEST"
)

pulumi.export("table_name", table.name)
```

Deploy with:

```
pulumi up
```

---

3. **Aurora**: High-performance relational database.

Introduction to Amazon Aurora

Amazon Aurora is a **fully managed relational database service** designed for high performance, scalability, and availability. It is compatible with **MySQL** and **PostgreSQL**, offering **five times the performance of MySQL** and **three times the performance of PostgreSQL**. Aurora automatically replicates data across **multiple Availability Zones (AZs)** and provides **continuous backups to Amazon S3**.

---

1. Configure Aurora via AWS UI (Console)

1. **Go to AWS Console** → Navigate to **RDS**

2. Click **Create database** → Select **Amazon Aurora**
3. Choose **Engine Type** (MySQL or PostgreSQL)
4. Configure **DB Instance Class**, Storage, and Connectivity
5. Enable **Multi-AZ Deployment** for high availability
6. Set up authentication and additional configurations
7. Click **Create Database**

---

2. Configure Aurora via AWS CLI

Create an Aurora Cluster

```
aws rds create-db-cluster \
  --db-cluster-identifier my-aurora-cluster \
  --engine aurora-mysql \
  --master-username admin \
  --master-user-password MySecurePass \
  --region us-east-1
```

Create an Aurora Instance

```
aws rds create-db-instance \
  --db-instance-identifier my-aurora-instance \
  --db-cluster-identifier my-aurora-cluster \
  --engine aurora-mysql \
  --db-instance-class db.r6g.large
```

---

3. Configure Aurora via AWS CloudFormation

Aurora CloudFormation Template

yaml

```
AWSTemplateFormatVersion: "2010-09-09"
```

```
Resources:
  AuroraDBCluster:
    Type: "AWS::RDS::DBCluster"
    Properties:
      Engine: "aurora-mysql"
      DBClusterIdentifier: "my-aurora-cluster"
      MasterUsername: "admin"
      MasterUserPassword: "MySecurePass"
      StorageEncrypted: true
  AuroraDBInstance:
    Type: "AWS::RDS::DBInstance"
    Properties:
      Engine: "aurora-mysql"
      DBClusterIdentifier: !Ref AuroraDBCluster
      DBInstanceClass: "db.r6g.large"
```

---

4. Configure Aurora via Terraform

Terraform Configuration for Aurora

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_rds_cluster" "aurora" {
  cluster_identifier = "my-aurora-cluster"
  engine             = "aurora-mysql"
  master_username    = "admin"
  master_password    = "MySecurePass"
}
```

```
resource "aws_rds_cluster_instance" "aurora_instance" {
  cluster_identifier = aws_rds_cluster.aurora.id
  instance_class     = "db.r6g.large"
  engine             = "aurora-mysql"
}
```

---

5. Configure Aurora via Pulumi

Pulumi Configuration for Aurora (Python)

python

```
import pulumi
import pulumi_aws as aws

aurora_cluster = aws.rds.Cluster("myAuroraCluster",
    engine="aurora-mysql",
    master_username="admin",
    master_password="MySecurePass")

aurora_instance = aws.rds.ClusterInstance("myAuroraInstance",
    cluster_identifier=aurora_cluster.id,
    instance_class="db.r6g.large",
    engine="aurora-mysql")

pulumi.export("aurora_cluster_id", aurora_cluster.id)
```

---

3. **Redshift**: Data warehousing.

Amazon Redshift: Data Warehousing

Amazon Redshift is a fully managed, petabyte-scale data warehouse service in AWS. It enables fast query performance using columnar storage and parallel query execution, making it ideal for analytical workloads. Redshift integrates with various AWS services like S3, Glue, and Athena for seamless data processing.

Configuring Amazon Redshift

1. Using AWS UI (Console)

- Navigate to the **AWS Management Console → Redshift**.
- Click **Create cluster** and choose the cluster type (RA3, DC2, etc.).
- Configure the node type, number of nodes, database name, and master credentials.
- Select **VPC, security group**, and configure backup settings.
- Click **Create cluster** to launch the Redshift instance.

---

2. Using AWS CLI

```
aws redshift create-cluster \
  --cluster-identifier my-redshift-cluster \
  --node-type dc2.large \
  --master-username admin \
  --master-user-password MyPassword123 \
  --number-of-nodes 2 \
  --region us-east-1
```

- Replace dc2.large with the required node type.
- Modify number-of-nodes as per the requirement.

---

3. Using AWS CloudFormation

yaml

```
Resources:
  RedshiftCluster:
    Type: AWS::Redshift::Cluster
    Properties:
      ClusterIdentifier: my-redshift-cluster
      NodeType: dc2.large
      MasterUsername: admin
      MasterUserPassword: MyPassword123
      NumberOfNodes: 2
      PubliclyAccessible: false
```

Save this as redshift.yaml and deploy with:

```
aws cloudformation create-stack --stack-name redshift-stack --template-body file://redshift.yaml
```

---

4. Using Terraform

```
hcl

provider "aws" {
  region = "us-east-1"
}

resource "aws_redshift_cluster" "example" {
  cluster_identifier = "my-redshift-cluster"
  database_name      = "mydatabase"
  master_username    = "admin"
  master_password    = "MyPassword123"
  node_type          = "dc2.large"
  cluster_type       = "multi-node"
  number_of_nodes    = 2
}
```

Run:
terraform init
terraform apply --auto-approve

---

5. Using Pulumi (Python)

python

```
import pulumi
import pulumi_aws as aws

redshift_cluster = aws.redshift.Cluster("myRedshiftCluster",
    cluster_identifier="my-redshift-cluster",
    node_type="dc2.large",
    master_username="admin",
    master_password="MyPassword123",
    number_of_nodes=2)

pulumi.export("redshift_endpoint", redshift_cluster.endpoint)
```

Run:
pulumi up

---

4. ElastiCache: In-memory caching (Redis/Memcached).

Amazon ElastiCache Introduction

Amazon ElastiCache is a fully managed in-memory caching service that supports **Redis** and **Memcached**. It enhances application performance by reducing database load and improving response times for read-heavy workloads. It integrates seamlessly with AWS services, providing low-latency data access.

ElastiCache Configuration Methods

1. AWS Management Console (UI)

1. **Navigate to ElastiCache**: Open the AWS Console and go to **ElastiCache**.
2. **Create Cluster**:
    - Select **"Create ElastiCache Cluster"**.
    - Choose **Redis** or **Memcached**.
    - Configure **node type, number of nodes, and security settings**.
3. **Set Security and Access**:
    - Attach a security group.
    - Enable encryption if needed.
4. **Launch Cluster** and wait for it to be **available**.
5. **Connect** using the provided **endpoint**.

---

2. AWS CLI

Create a Redis Cluster

```
aws elasticache create-cache-cluster \
  --cache-cluster-id my-redis-cluster \
  --engine redis \
  --cache-node-type cache.t3.micro \
  --num-cache-nodes 1 \
  --security-group-ids sg-xxxxxxxx \
  --cache-subnet-group-name my-subnet-group
```

Create a Memcached Cluster

```
aws elasticache create-cache-cluster \
  --cache-cluster-id my-memcached-cluster \
  --engine memcached \
  --cache-node-type cache.t3.micro \
  --num-cache-nodes 2 \
  --security-group-ids sg-xxxxxxxx
```

List Clusters

```
aws elasticache describe-cache-clusters
```

---

3. AWS CloudFormation

Redis Cluster Configuration

yaml

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  MyElastiCacheCluster:
    Type: "AWS::ElastiCache::CacheCluster"
    Properties:
      CacheClusterId: "my-redis-cluster"
      Engine: "redis"
      CacheNodeType: "cache.t3.micro"
      NumCacheNodes: 1
      VpcSecurityGroupIds:
        - "sg-xxxxxxxx"
```

---

4. Terraform Configuration

Redis Cluster

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_elasticache_cluster" "redis" {
  cluster_id           = "my-redis-cluster"
  engine               = "redis"
  node_type            = "cache.t3.micro"
  num_cache_nodes      = 1
  parameter_group_name = "default.redis7"
  security_group_ids   = ["sg-xxxxxxxx"]
}
```

---

5. Pulumi Configuration (TypeScript)

Redis Cluster

typescript

```
import * as aws from "@pulumi/aws";

const redisCluster = new aws.elasticache.Cluster("redisCluster", {
    clusterId: "my-redis-cluster",
    engine: "redis",
    nodeType: "cache.t3.micro",
    numCacheNodes: 1,
    parameterGroupName: "default.redis7",
    securityGroupIds: ["sg-xxxxxxxx"]
});

export const clusterEndpoint = redisCluster.cacheNodes[0].address;
```

---

5. **Neptune**: Graph database.

Introduction to AWS Neptune: Graph Database

Amazon Neptune is a managed graph database service designed for highly connected data, supporting both **property graph** (Gremlin) and **RDF (SPARQL)** models. It is ideal for use cases like fraud detection, recommendation engines, and knowledge graphs. Neptune is fully managed, highly available, and supports automatic backups and read replicas for scalability.

---

1. AWS Management Console (UI)

1. Open the **AWS Console** and navigate to **Amazon Neptune**.
2. Click **Create database** and choose **Engine type** (Gremlin or SPARQL).
3. Select **DB instance size**, storage, and VPC settings.
4. Enable encryption, backup retention, and IAM authentication if needed.
5. Click **Create database** and wait for the instance to become available.

---

2. AWS CLI Configuration

Create an Amazon Neptune cluster using AWS CLI:

```
aws neptune create-db-cluster \
  --db-cluster-identifier my-neptune-cluster \
  --engine neptune \
  --master-username admin \
  --master-user-password MyPassword123 \
  --region us-east-1
```

Create a Neptune DB instance:

```
aws neptune create-db-instance \
    --db-instance-identifier my-neptune-instance \
    --db-cluster-identifier my-neptune-cluster \
    --instance-class db.t3.medium \
    --engine neptune
```

---

3. AWS CloudFormation Configuration

Deploy Neptune using a CloudFormation template:

```
Resources:
  NeptuneDBCluster:
    Type: "AWS::Neptune::DBCluster"
    Properties:
      DBClusterIdentifier: "my-neptune-cluster"
      Engine: "neptune"
      MasterUsername: "admin"
      MasterUserPassword: "MyPassword123"

  NeptuneDBInstance:
    Type: "AWS::Neptune::DBInstance"
    Properties:
      DBInstanceIdentifier: "my-neptune-instance"
      DBClusterIdentifier: !Ref NeptuneDBCluster
      DBInstanceClass: "db.t3.medium"
      Engine: "neptune"
```

Deploy using AWS CLI:

```
aws cloudformation create-stack --stack-name my-neptune-stack --template-body file://neptune.yaml
```

---

4. Terraform Configuration

Create a Neptune database cluster using Terraform:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_neptune_cluster" "example" {
  cluster_identifier   = "my-neptune-cluster"
  engine               = "neptune"
  master_username      = "admin"
  master_password      = "MyPassword123"
}

resource "aws_neptune_cluster_instance" "example" {
  count              = 1
  cluster_identifier  = aws_neptune_cluster.example.id
  instance_class      = "db.t3.medium"
  engine             = "neptune"
}
```

Apply Terraform:

```
terraform init
terraform apply -auto-approve
```

---

5. Pulumi Configuration (Python)

Create a Neptune cluster using Pulumi:

python

```
import pulumi
import pulumi_aws as aws

neptune_cluster = aws.neptune.Cluster(
    "myNeptuneCluster",
    cluster_identifier="my-neptune-cluster",
    engine="neptune",
    master_username="admin",
    master_password="MyPassword123",
)

neptune_instance = aws.neptune.ClusterInstance(
    "myNeptuneInstance",
    cluster_identifier=neptune_cluster.id,
    instance_class="db.t3.medium",
    engine="neptune",
)
```

Deploy with Pulumi:

```
pulumi up
```

---

6. **DocumentDB**: Managed NoSQL document database.

Introduction

Amazon DocumentDB is a fully managed NoSQL document database service designed for JSON-based workloads. It is compatible with MongoDB, allowing users to store, query, and index JSON data efficiently. DocumentDB provides high

availability, automatic backups, and scalability while offloading infrastructure management.

---

1. AWS UI Configuration

1. **Login to AWS Console** and navigate to **Amazon DocumentDB**.
2. Click **Create Cluster** and configure:
    - Cluster name
    - Instance class
    - Number of instances
    - Authentication and security settings
3. **Enable backups and encryption** as needed.
4. Click **Create Cluster** and wait for the cluster to be provisioned.
5. Use the **MongoDB shell or AWS SDKs** to connect to the cluster.

---

2. AWS CLI Configuration

Create a cluster:

```
aws docdb create-db-cluster \
  --db-cluster-identifier my-docdb-cluster \
  --engine docdb \
  --master-username admin \
  --master-user-password MySecurePassword
```

Create an instance:

```
aws docdb create-db-instance \
  --db-instance-identifier my-docdb-instance \
  --db-instance-class db.r5.large \
  --engine docdb \
  --db-cluster-identifier my-docdb-cluster
```

Get the cluster endpoint:
aws docdb describe-db-clusters --query "DBClusters[*].Endpoint"

---

3. CloudFormation Configuration

Example CloudFormation template:

yaml

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  DocumentDBCluster:
    Type: AWS::DocDB::DBCluster
    Properties:
      DBClusterIdentifier: "MyDocDBCluster"
      MasterUsername: "admin"
      MasterUserPassword: "MySecurePassword"
      EngineVersion: "4.0"
      BackupRetentionPeriod: 7
```

Deploy using:

aws cloudformation create-stack --stack-name my-docdb-stack --template-body file://docdb.yaml

---

4. Terraform Configuration

Terraform example:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_docdb_cluster" "example" {
  cluster_identifier      = "my-docdb-cluster"
  master_username         = "admin"
  master_password         = "MySecurePassword"
  backup_retention_period = 7
}

resource "aws_docdb_cluster_instance" "example" {
  identifier         = "my-docdb-instance"
  cluster_identifier = aws_docdb_cluster.example.id
  instance_class     = "db.r5.large"
}
```

Deploy using:

```
terraform init
terraform apply
```

---

5. Pulumi Configuration

Pulumi (Python) example:

```
import pulumi
import pulumi_aws as aws

cluster = aws.docdb.Cluster("myDocDBCluster",
    master_username="admin",
    master_password="MySecurePassword",
    backup_retention_period=7
```

```
)

instance = aws.docdb.ClusterInstance("myDocDBInstance",
    cluster_identifier=cluster.id,
    instance_class="db.r5.large"
)

pulumi.export("endpoint", cluster.endpoint)
```

Deploy using:

```
pulumi up
```

---

7. **Timestream**: Time-series database.

Introduction to AWS Timestream

AWS Timestream is a fast, scalable, and serverless time-series database service designed for IoT applications, operational analytics, and real-time monitoring. It automatically scales, handles data lifecycle management, and supports SQL-based queries optimized for time-series data.

---

1. AWS UI Configuration

1. Open **AWS Management Console**
2. Navigate to **Timestream** service
3. Click **Create database**
4. Choose **Standard database** or **Sample database**
5. Define **Database name** and select **KMS encryption** (optional)

6. Click **Create database**
7. Create a table under the database and define retention policies

---

2. AWS CLI Configuration

Create a Timestream Database

aws timestream-write create-database --database-name my-timestream-db

Create a Table with Retention Policies

```
aws timestream-write create-table --database-name my-timestream-db \
  --table-name my-table \
  --retention-properties
MemoryStoreRetentionPeriodInHours=24,MagneticStoreRetentionPeriodInDays=
7
```

List Databases

aws timestream-write list-databases

---

3. AWS CloudFormation Configuration

Timestream Database and Table

yaml

```
Resources:
  TimestreamDatabase:
    Type: AWS::Timestream::Database
    Properties:
      DatabaseName: MyTimestreamDB

  TimestreamTable:
```

```
  Type: AWS::Timestream::Table
  Properties:
    DatabaseName: !Ref TimestreamDatabase
    TableName: MyTimestreamTable
    RetentionProperties:
      MemoryStoreRetentionPeriodInHours: 24
      MagneticStoreRetentionPeriodInDays: 7
```

---

4. Terraform Configuration

Timestream Database and Table

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_timestreamwrite_database" "example" {
  database_name = "my_timestream_db"
}

resource "aws_timestreamwrite_table" "example" {
  database_name = aws_timestreamwrite_database.example.database_name
  table_name    = "my_timestream_table"

  retention_properties {
    memory_store_retention_period_in_hours  = 24
    magnetic_store_retention_period_in_days = 7
  }
}
```

---

5. Pulumi Configuration (Python)

Timestream Database and Table
python

```
import pulumi
import pulumi_aws as aws

timestream_db = aws.timestreamwrite.Database("myTimestreamDB")

timestream_table = aws.timestreamwrite.Table(
    "myTimestreamTable",
    database_name=timestream_db.database_name,
    retention_properties={
        "memory_store_retention_period_in_hours": 24,
        "magnetic_store_retention_period_in_days": 7
    }
)

pulumi.export("timestream_database", timestream_db.database_name)
pulumi.export("timestream_table", timestream_table.table_name)
```

---

8. Keyspaces: Managed Cassandra-compatible database.

AWS Keyspaces (Managed Cassandra-compatible database) is a scalable, highly available, and managed database service designed to handle workloads requiring Apache Cassandra compatibility. It offers the benefits of a NoSQL database with ease of use, providing automatic scaling, fault tolerance, and integration with other AWS services.

---

1. AWS UI (Console) Configuration:

- Navigate to the **AWS Keyspaces** service in the AWS Management Console.
- Click on **Create keyspace**.
- Provide a name for the keyspace and configure the settings like replication factor, and choose the **AWS region** where you want the database to be deployed.
- Once configured, click **Create**.

---

2. CLI Configuration:

AWS CLI allows you to interact with Keyspaces via the command line.
Example:

aws keyspaces create-keyspace --keyspace-name my_keyspace

- This command creates a new keyspace called my_keyspace.
- You can also configure replication settings, schemas, and more using various commands.

---

3. CloudFormation Configuration:

AWS CloudFormation allows you to define Keyspaces resources in a template for automatic deployment. Example:

yaml

```
Resources:
  MyKeyspace:
    Type: AWS::Keyspaces::Keyspace
```

```
  Properties:
    KeyspaceName: my_keyspace
```

- Use this YAML template to create a new keyspace in your CloudFormation stack.

---

4. Terraform Configuration:

Terraform is an infrastructure-as-code tool for automating AWS resources, including Keyspaces. Example:

hcl

```
resource "aws_keyspaces_keyspace" "my_keyspace" {
  keyspace_name = "my_keyspace"
}
```

- This code defines a Keyspace resource in Terraform, allowing it to be managed and automated with version-controlled infrastructure code.

---

5. Pulumi Configuration:

Pulumi is another IaC tool for managing cloud resources. Here's how to configure Keyspaces: Example:

javascript

```
const keyspaces = require("@pulumi/aws/keyspaces");
```

```
const myKeyspace = new keyspaces.Keyspace("my_keyspace", {
   keyspaceName: "my_keyspace",
});
```

- This code configures a Keyspace using Pulumi's AWS SDK for managing AWS Keyspaces resources.

Networking & Content Delivery

1. VPC (Virtual Private Cloud): Isolated network for AWS resources.

Introduction to VPC (Virtual Private Cloud)

A **Virtual Private Cloud (VPC)** is an isolated network within AWS where you can define and launch AWS resources. VPC gives you full control over network settings like IP address ranges, subnets, routing tables, and network gateways. It helps you securely connect your resources and control inbound and outbound traffic to/from the internet.

---

1. AWS UI (Console)

1. **Log in to AWS Console**: Navigate to the VPC dashboard.
2. **Create VPC**:
    - Go to VPC > Create VPC.
    - Define CIDR block (e.g., 10.0.0.0/16).
    - Select the type of VPC (default or custom).
    - Click Create.
3. **Create Subnets**: After creating the VPC, create public and private subnets within it.

4. **Set up Route Tables**: Configure routing for internet access or peering connections.

---

2. AWS CLI

Create VPC:
aws ec2 create-vpc --cidr-block 10.0.0.0/16 --region us-east-1

Create Subnet:
aws ec2 create-subnet --vpc-id vpc-xxxxxx --cidr-block 10.0.1.0/24 --availability-zone us-east-1a

Create Internet Gateway and attach it:
aws ec2 create-internet-gateway
aws ec2 attach-internet-gateway --vpc-id vpc-xxxxxx --internet-gateway-id igw-xxxxxx

---

3. CloudFormation

CloudFormation allows you to define your infrastructure as code. Below is an example template to create a VPC:

yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  MyVPC:
    Type: AWS::EC2::VPC
```

```
    Properties:
      CidrBlock: "10.0.0.0/16"
      EnableDnsSupport: "true"
      EnableDnsHostnames: "true"

  MySubnet:
    Type: AWS::EC2::Subnet
    Properties:
      VpcId: !Ref MyVPC
      CidrBlock: "10.0.1.0/24"
      AvailabilityZone: "us-east-1a"
```

To deploy:

```
aws cloudformation create-stack --stack-name my-vpc-stack --template-body file://vpc-template.yaml
```

---

4. Terraform

With Terraform, you can automate the creation of the VPC:

```
hcl

provider "aws" {
  region = "us-east-1"
}

resource "aws_vpc" "main" {
  cidr_block = "10.0.0.0/16"
  enable_dns_support = true
  enable_dns_hostnames = true
}
```

```
resource "aws_subnet" "subnet" {
  vpc_id = aws_vpc.main.id
  cidr_block = "10.0.1.0/24"
  availability_zone = "us-east-1a"
}
```

To apply the configuration:

```
terraform init
terraform apply
```

---

5. Pulumi

With Pulumi, you can create a VPC in Python, TypeScript, or Go. Here's an example using Python:

```
python

import pulumi
import pulumi_aws as aws

vpc = aws.ec2.Vpc('my-vpc', cidr_block='10.0.0.0/16')
subnet = aws.ec2.Subnet('my-subnet', cidr_block='10.0.1.0/24', vpc_id=vpc.id)
```

To deploy:

```
pulumi up
```

---

2. CloudFront: Content delivery network (CDN).

AWS CloudFront: Content Delivery Network (CDN)

AWS CloudFront is a global content delivery network (CDN) service that accelerates the delivery of your websites, APIs, video content, and other web assets to users worldwide. By caching content at edge locations, CloudFront reduces latency and improves download speeds.

---

1. AWS Management Console (UI)

To configure CloudFront using the AWS UI:

1. Go to the AWS CloudFront Console.
2. Click **Create Distribution**.
3. Choose the delivery method (Web or RTMP).
4. Under **Origin Settings**, set the **Origin Domain Name** (e.g., S3 bucket or custom server).
5. Configure caching behavior, SSL, and other settings.
6. Click **Create Distribution**.

---

2. AWS CLI Configuration

To create a CloudFront distribution via AWS CLI, you can use the following command:

```
aws cloudfront create-distribution \
  --origin-domain-name example-bucket.s3.amazonaws.com \
  --default-root-object index.html \
  --enabled
```

You can customize options like SSL certificates, cache behaviors, and logging by adding additional parameters.

---

3. AWS CloudFormation

Using AWS CloudFormation, you can define your CloudFront distribution in a YAML or JSON template. Below is an example YAML configuration for CloudFront:

yaml

```
Resources:
  MyCloudFrontDistribution:
    Type: AWS::CloudFront::Distribution
    Properties:
      DistributionConfig:
        Origins:
          - Id: MyS3Origin
            DomainName: example-bucket.s3.amazonaws.com
            S3OriginConfig: {}
        DefaultCacheBehavior:
          TargetOriginId: MyS3Origin
          ViewerProtocolPolicy: redirect-to-https
        Enabled: true
```

Use aws cloudformation create-stack to deploy the configuration.

---

4. Terraform Configuration

With Terraform, you can define CloudFront distributions as follows:

```
resource "aws_cloudfront_distribution" "example" {
  origin {
    domain_name = "example-bucket.s3.amazonaws.com"
    origin_id   = "S3-origin"
```

```
  }

  enabled             = true
  is_ipv6_enabled     = true
  default_root_object = "index.html"

  default_cache_behavior {
    target_origin_id       = "S3-origin"
    viewer_protocol_policy = "redirect-to-https"
  }
}
```

Apply the configuration using terraform apply.

---

5. Pulumi Configuration

In Pulumi, you can configure CloudFront as shown below using TypeScript:

typescript

```
import * as aws from "@pulumi/aws";

const distribution = new aws.cloudfront.Distribution("my-cdn", {
    origins: [{
        domainName: "example-bucket.s3.amazonaws.com",
        originId: "S3-origin",
    }],
    enabled: true,
    defaultRootObject: "index.html",
    defaultCacheBehavior: {
        targetOriginId: "S3-origin",
        viewerProtocolPolicy: "redirect-to-https",
    },
```

});

Run pulumi up to deploy this configuration.

---

3. Route 53: Domain Name System (DNS) service.

Introduction to AWS Route 53

AWS Route 53 is a scalable and highly available Domain Name System (DNS) web service designed to route end users to Internet applications by translating friendly domain names like www.example.com into IP addresses. It offers three main functionalities:

- **DNS Management**: Managing domain names and routing traffic.
- **Domain Registration**: Registering new domain names.
- **Health Checking**: Monitoring the health of resources.

Route 53 Configuration Methods

1. AWS UI (Console)

1. Go to the AWS Route 53 Console.
2. Click on **Create Hosted Zone**.
3. Enter the domain name (e.g., example.com) and select the type of hosted zone (Public or Private).
4. After the zone is created, you can add Record Sets (A, CNAME, etc.) to route traffic to your desired endpoints (e.g., EC2 instances, S3, etc.).

---

2. AWS CLI

To configure Route 53 using the AWS CLI, you can use the following commands:

Create Hosted Zone:
aws route53 create-hosted-zone --name example.com --caller-reference unique-id

Create Record Set (example for an A record):

```
aws route53 change-resource-record-sets --hosted-zone-id Z3M3LMEXAMPLE
--change-batch '{
  "Changes": [{
    "Action": "CREATE",
    "ResourceRecordSet": {
      "Name": "www.example.com.",
      "Type": "A",
      "TTL": 60,
      "ResourceRecords": [{"Value": "192.0.2.44"}]
    }
  }]
}'
```

---

3. CloudFormation

AWS CloudFormation allows you to define Route 53 configurations as part of your infrastructure as code. Below is a CloudFormation template to create a hosted zone and a record set:

yaml

```
Resources:
  MyHostedZone:
```

```
    Type: "AWS::Route53::HostedZone"
    Properties:
      Name: "example.com"

  MyRecordSet:
    Type: "AWS::Route53::RecordSet"
    Properties:
      HostedZoneId: !Ref MyHostedZone
      Name: "www.example.com"
      Type: "A"
      TTL: "60"
      ResourceRecords:
        - "192.0.2.44"
```

---

4. Terraform

Terraform allows you to manage Route 53 resources with infrastructure as code. Below is a basic Terraform configuration for Route 53:

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_route53_zone" "example" {
  name = "example.com"
}

resource "aws_route53_record" "www" {
  zone_id = aws_route53_zone.example.id
  name    = "www.example.com"
  type    = "A"
```

```
  ttl     = 60
  records = ["192.0.2.44"]
}
```

---

5. Pulumi

Pulumi also allows you to define infrastructure as code in multiple languages (JavaScript, TypeScript, Python, etc.). Below is an example of how you can configure Route 53 in Pulumi using Python:

python

```
import pulumi
import pulumi_aws as aws

zone = aws.route53.Zone("example", name="example.com")

record = aws.route53.Record("www",
    zone_id=zone.id,
    name="www.example.com",
    type="A",
    ttl=60,
    records=["192.0.2.44"])
```

---

4. Direct Connect: Dedicated network connection to AWS.

AWS Direct Connect is a cloud service that establishes a dedicated network connection from your premises to AWS. It provides more reliable, consistent performance and lower latencies compared to typical internet connections. You can use Direct Connect for various purposes like accessing AWS services, using hybrid cloud configurations, or improving data transfer speeds.

---

1. AWS Management Console (UI)

To configure AWS Direct Connect through the AWS Management Console:

- Go to the **Direct Connect** section in the AWS console.
- Click on **Create Connection**.
- Select your AWS region and specify the **connection details** like connection name, bandwidth, and location (e.g., AWS Direct Connect location or colocation facility).
- After creation, you'll configure the **virtual interface** to connect to your VPC or other AWS resources.

---

2. AWS CLI

Using the AWS CLI, you can create and manage Direct Connect connections as follows:

```
aws directconnect create-connection \
  --location "New York" \
  --bandwidth "1Gbps" \
  --connection-name "MyDirectConnectConnection"
```

To create a virtual interface after the connection:

```
aws directconnect create-private-virtual-interface \
  --connection-id dxcon-xxxx \
  --new-private-virtual-interface '{"virtualInterfaceName": "MyVIF", "vlan": 101, "asn": 65000, "authKey": "myAuthKey", "addressFamily": "ipv4", "amazonAddress": "192.168.1.1", "customerAddress": "192.168.1.2"}'
```

---

3. CloudFormation

CloudFormation templates can be used to automate Direct Connect configuration. Here's an example snippet for defining a Direct Connect connection:

yaml

```
Resources:
  MyDirectConnectConnection:
    Type: AWS::DirectConnect::Connection
    Properties:
      Location: "New York"
      Bandwidth: "1Gbps"
      ConnectionName: "MyDirectConnectConnection"
  MyPrivateVirtualInterface:
    Type: AWS::DirectConnect::VirtualInterface
    Properties:
      ConnectionId: !Ref MyDirectConnectConnection
      VirtualInterfaceName: "MyVIF"
      Vlan: 101
      Asn: 65000
      AmazonAddress: "192.168.1.1"
      CustomerAddress: "192.168.1.2"
      AddressFamily: "ipv4"
      AuthKey: "myAuthKey"
```

---

4. Terraform

In Terraform, you can configure Direct Connect like this:

hcl

```
resource "aws_dx_connection" "example" {
  name      = "MyDirectConnectConnection"
```

```
  location   = "New York"
  bandwidth  = "1Gbps"
}

resource "aws_dx_private_virtual_interface" "example" {
  connection_id       = aws_dx_connection.example.id
  name                = "MyVIF"
  vlan                = 101
  asn                 = 65000
  amazon_address      = "192.168.1.1"
  customer_address    = "192.168.1.2"
  address_family      = "ipv4"
  auth_key            = "myAuthKey"
}
```

---

5. Pulumi

With Pulumi, you can define the infrastructure in your preferred language. Here's an example in TypeScript:

typescript

```
import * as aws from "@pulumi/aws";

// Create a Direct Connect connection
const connection = new aws.directconnect.Connection("myConnection", {
    location: "New York",
    bandwidth: "1Gbps",
    connectionName: "MyDirectConnectConnection",
});

// Create a virtual interface
const virtualInterface = new aws.directconnect.PrivateVirtualInterface("myVIF", {
```

```
    connectionId: connection.id,
    virtualInterfaceName: "MyVIF",
    vlan: 101,
    asn: 65000,
    amazonAddress: "192.168.1.1",
    customerAddress: "192.168.1.2",
    addressFamily: "ipv4",
    authKey: "myAuthKey",
});
```

---

5. Global Accelerator: Improves application performance.

AWS Global Accelerator:

AWS Global Accelerator improves the performance of your applications by directing traffic to the optimal endpoint based on health, geography, and routing policies. It uses AWS's global network infrastructure to route user traffic, reducing latency and improving availability.

---

1. Configuration via AWS UI (Console):

1. **Navigate to the AWS Global Accelerator service** in the AWS Management Console.
2. **Create an accelerator**: Click "Create accelerator" and enter a name.
3. **Define listeners**: Choose a protocol (TCP/UDP) and port range for the listener.
4. **Add endpoints**: Select the regions and resources (like EC2 instances, load balancers) that will act as endpoints.
5. **Review and Create**: Review the configuration and create the accelerator.

---

2. Configuration via AWS CLI:

Using AWS CLI, you can configure Global Accelerator with commands like:

aws globalaccelerator create-accelerator --name my-accelerator --ip-address-type IPV4
aws globalaccelerator create-listener --accelerator-arn <accelerator-arn> --port-range 80 --protocol TCP
aws globalaccelerator create-endpoint-group --listener-arn <listener-arn> --endpoint-configurations <endpoint-configurations>

This helps automate the process in scripting or CI/CD pipelines.

---

3. Configuration via CloudFormation:

CloudFormation allows you to define Global Accelerator configurations as code. Here's an example template:

yaml

```
Resources:
  MyGlobalAccelerator:
    Type: AWS::GlobalAccelerator::Accelerator
    Properties:
      Name: "my-accelerator"
      IpAddressType: "IPV4"

  MyGlobalListener:
    Type: AWS::GlobalAccelerator::Listener
    Properties:
      AcceleratorArn: !Ref MyGlobalAccelerator
      PortRanges:
```

```
      - FromPort: 80
        ToPort: 80
    Protocol: "TCP"
```

This CloudFormation stack can be deployed to set up Global Accelerator.

---

4. Configuration via Terraform:

Using Terraform, Global Accelerator can be configured as follows:

hcl

```
resource "aws_globalaccelerator_accelerator" "example" {
  name            = "my-accelerator"
  ip_address_type  = "IPV4"
  enabled         = true
}

resource "aws_globalaccelerator_listener" "example" {
  accelerator_arn = aws_globalaccelerator_accelerator.example.id
  protocol        = "TCP"
  port_range {
    from_port = 80
    to_port   = 80
  }
}

resource "aws_globalaccelerator_endpoint_group" "example" {
  listener_arn = aws_globalaccelerator_listener.example.id
  endpoint_configuration {
    endpoint_id   = "endpoint-id"
    weight        = 128
  }
```

```
}
```

This Terraform configuration sets up a Global Accelerator, listener, and endpoint group.

---

5. Configuration via Pulumi:

Pulumi allows you to configure Global Accelerator in code using a programming language like JavaScript, TypeScript, or Python. Here's an example in TypeScript:

typescript

```
import * as aws from "@pulumi/aws";

const accelerator = new aws.globalaccelerator.Accelerator("myAccelerator", {
    ipAddressType: "IPV4",
});

const listener = new aws.globalaccelerator.Listener("myListener", {
    acceleratorArn: accelerator.arn,
    portRanges: [{ fromPort: 80, toPort: 80 }],
    protocol: "TCP",
});

const endpointGroup = new
aws.globalaccelerator.EndpointGroup("myEndpointGroup", {
    listenerArn: listener.arn,
    endpointConfigurations: [{
        endpointId: "my-endpoint-id",
        weight: 128,
    }],
});
```

This Pulumi configuration defines the Global Accelerator, listener, and endpoints programmatically.

---

6. Elastic Load Balancing (ELB): Distributes incoming traffic.

Elastic Load Balancing (ELB)

Elastic Load Balancing (ELB) automatically distributes incoming traffic across multiple targets such as EC2 instances, containers, and IP addresses. ELB helps ensure that no single instance bears too much load, leading to better application performance and reliability.

---

1. AWS UI (Console)

1. **Log in** to the AWS Management Console.
2. Navigate to **EC2 Dashboard** > **Load Balancers**.
3. Click **Create Load Balancer**.
4. Select the desired load balancer type (Application, Network, or Classic).
5. Configure the load balancer, including the name, scheme (internet-facing or internal), and listeners.
6. Select **Availability Zones** and associate **targets** (EC2 instances).
7. Configure health checks, security groups, and tags.
8. Review and create the ELB.

---

2. AWS CLI

To create an Application Load Balancer (ALB) using AWS CLI:

aws elb create-load-balancer --load-balancer-name my-load-balancer \

```
    --listeners
Protocol=HTTP,LoadBalancerPort=80,InstanceProtocol=HTTP,InstancePort=80 \
    --subnets subnet-12345678 subnet-87654321 \
    --security-groups sg-12345678
```

You can further configure health checks, target groups, and more using the AWS CLI.

---

3. CloudFormation

A sample CloudFormation template to create an ALB:

yaml

```
Resources:
  MyLoadBalancer:
    Type: 'AWS::ElasticLoadBalancingV2::LoadBalancer'
    Properties:
      Name: my-load-balancer
      Subnets:
        - subnet-12345678
        - subnet-87654321
      SecurityGroups:
        - sg-12345678
      Scheme: internet-facing
      LoadBalancerAttributes:
        - Key: idle_timeout.timeout_seconds
          Value: '60'
      Type: application
```

This template will create an Application Load Balancer in the specified subnets with security groups.

---

4. Terraform

To create an ELB using Terraform:

hcl

```
resource "aws_lb" "my_lb" {
  name               = "my-load-balancer"
  internal           = false
  load_balancer_type = "application"
  security_groups    = ["sg-12345678"]
  subnets            = ["subnet-12345678", "subnet-87654321"]

  enable_deletion_protection = false
}

resource "aws_lb_target_group" "my_target_group" {
  name     = "my-target-group"
  port     = 80
  protocol = "HTTP"
  vpc_id   = "vpc-12345678"
}

resource "aws_lb_listener" "my_listener" {
  load_balancer_arn = aws_lb.my_lb.arn
  port              = "80"
  protocol          = "HTTP"

  default_action {
    type             = "fixed-response"
    fixed_response {
      status_code = 200
      content_type = "text/plain"
```

```
    message_body = "OK"
  }
 }
}
```

---

5. Pulumi

Here's an example of creating an ALB with Pulumi in TypeScript:

typescript

```
import * as aws from "@pulumi/aws";

// Create an Application Load Balancer
const lb = new aws.lb.LoadBalancer("my-load-balancer", {
  internal: false,
  loadBalancerType: "application",
  securityGroups: ["sg-12345678"],
  subnets: ["subnet-12345678", "subnet-87654321"],
});

// Create a Target Group
const targetGroup = new aws.lb.TargetGroup("my-target-group", {
  port: 80,
  protocol: "HTTP",
  vpcId: "vpc-12345678",
});

// Create a Listener for HTTP
const listener = new aws.lb.Listener("my-listener", {
  loadBalancerArn: lb.arn,
  port: 80,
  protocol: "HTTP",
```

```
    defaultActions: [{
        type: "fixed-response",
        fixedResponse: {
            statusCode: 200,
            contentType: "text/plain",
            messageBody: "OK",
        },
    }],
});
```

---

7. App Mesh: Service mesh for microservices.

AWS App Mesh: Service Mesh for Microservices

AWS App Mesh is a service mesh that provides application-level networking to make it easy for your microservices to communicate with each other. It allows you to manage and monitor your microservices across multiple types of compute infrastructure such as EC2 instances, ECS, and EKS. App Mesh uses Envoy proxy to route traffic and provides features like traffic shaping, observability, and resilience.

Configuration

1. AWS Management Console (UI) Configuration:

- **Step 1:** Log in to the AWS Management Console.
- **Step 2:** Navigate to **App Mesh** under the "Networking & Content Delivery" section.
- **Step 3:** Create a new mesh by clicking on **Create Mesh**, give it a name, and configure other options.

- **Step 4:** Add virtual services and virtual nodes representing your microservices.
- **Step 5:** Define virtual routers and routes to manage traffic between your microservices.

---

2. AWS CLI Configuration:

Using the AWS CLI, you can create and configure App Mesh resources by running commands:

Create a mesh:
aws appmesh create-mesh --mesh-name my-mesh

Create a virtual node:
aws appmesh create-virtual-node --mesh-name my-mesh --virtual-node-name my-node --spec file://virtual-node-spec.json

Create a virtual service:
aws appmesh create-virtual-service --mesh-name my-mesh --virtual-service-name my-service.local --spec file://virtual-service-spec.json

---

3. AWS CloudFormation Configuration:

Using CloudFormation, you can define the AWS App Mesh resources in a YAML or JSON template:

yaml

```
Resources:
  MyMesh:
    Type: AWS::AppMesh::Mesh
    Properties:
      MeshName: my-mesh
  MyVirtualNode:
    Type: AWS::AppMesh::VirtualNode
    Properties:
      MeshName: !Ref MyMesh
      VirtualNodeName: my-node
      Spec:
        ServiceDiscovery:
          DNS:
            Hostname: my-service.local
```

Deploy the CloudFormation stack:

```
aws cloudformation create-stack --stack-name my-app-mesh-stack --template-body file://appmesh-template.yaml
```

---

4. Terraform Configuration:

You can configure AWS App Mesh with Terraform:

```
hcl

resource "aws_appmesh_mesh" "my_mesh" {
  name = "my-mesh"
}

resource "aws_appmesh_virtual_node" "my_virtual_node" {
  mesh_name = aws_appmesh_mesh.my_mesh.name
```

```
  name      = "my-node"

  spec {
    service_discovery {
      dns {
        hostname = "my-service.local"
      }
    }
  }
}
```

Deploy the configuration:

terraform init
terraform apply

---

5. Pulumi Configuration:

Using Pulumi, you can write the configuration in TypeScript, Python, Go, or C# to provision AWS App Mesh resources:

typescript

```
import * as aws from "@pulumi/aws";

const mesh = new aws.appmesh.Mesh("my-mesh");

const virtualNode = new aws.appmesh.VirtualNode("my-node", {
  meshName: mesh.name,
  spec: {
    serviceDiscovery: {
      dns: {
        hostname: "my-service.local",
```

```
    },
  },
 },
});
```

Deploy the configuration:

pulumi up

Developer Tools

1. CodeCommit: Version control (Git).

AWS CodeCommit Introduction:

AWS CodeCommit is a fully managed source control service that you can use to host secure and scalable Git repositories. It helps developers to collaborate on code and easily integrate with CI/CD pipelines. CodeCommit offers high availability and scalability while keeping your code secure.

AWS CodeCommit Configuration

1. AWS UI (Console):

- **Step 1**: Go to the **AWS Management Console**, then search for **CodeCommit**.
- **Step 2**: Click **Create repository**.
- **Step 3**: Name your repository and provide an optional description.
- **Step 4**: After creating, follow the instructions to connect to the repository using HTTPS or SSH and push your code.

---

2. AWS CLI Configuration:

- Install and configure the **AWS CLI** if you haven't already.

aws configure

- **To create a new repository:**

aws codecommit create-repository --repository-name MyRepository
--repository-description "My CodeCommit Repo"

- **To clone and push to your repository:**

git clone
https://git-codecommit.us-west-2.amazonaws.com/v1/repos/MyRepository

---

3. AWS CloudFormation Configuration:

- CloudFormation allows you to create and manage resources in AWS through templates. To configure CodeCommit:

yaml

```
Resources:
  MyRepository:
    Type: AWS::CodeCommit::Repository
    Properties:
      RepositoryName: "MyRepository"
      Description: "A CodeCommit repository"
```

- **Deploy the stack with:**

aws cloudformation create-stack --stack-name CodeCommitStack --template-body file://codecommit-template.yaml

---

4. Terraform Configuration:

- Terraform allows you to manage AWS CodeCommit repositories as part of your infrastructure as code.

hcl

```
provider "aws" {
  region = "us-west-2"
}

resource "aws_codecommit_repository" "my_repo" {
  repository_name = "MyRepository"
  description     = "A CodeCommit repository"
}
```

- **To apply the configuration:**

terraform init
terraform apply

---

5. Pulumi Configuration:

- Pulumi provides an infrastructure as code solution that supports multiple cloud providers. To configure AWS CodeCommit:

typescript

```
import * as aws from "@pulumi/aws";

const repository = new aws.codecommit.Repository("my-repository", {
   repositoryName: "MyRepository",
   description: "A CodeCommit repository",
});
```

- **To deploy the stack:**

```
pulumi up
```

---

2. CodeBuild: Build and test code.

Introduction to AWS CodeBuild:

AWS CodeBuild is a fully managed build service in the cloud. It allows developers to compile source code, run tests, and produce ready-to-deploy artifacts. CodeBuild supports various programming languages and can be integrated with AWS CodePipeline or other CI/CD tools.

---

1. Configuration via AWS UI:

1. **Sign in to AWS Management Console.**
2. **Navigate to CodeBuild:**
    - Go to **Services > Developer Tools > CodeBuild**.
3. **Create a new Project:**
    - Click **Create build project**.
    - Fill in the project details, including the source repository (e.g., GitHub, Bitbucket, etc.).

4. **Buildspec File:**
    - If using a buildspec file (buildspec.yml), AWS CodeBuild automatically looks for it in the root of your project. Alternatively, you can define commands directly in the UI.
5. **Configure Environment:**
    - Select the operating system, runtime, and image for your build environment.
    - Set up build and artifact output settings.
6. **Build and Test:**
    - Start the build process, view logs, and monitor the status.

---

2. Configuration via AWS CLI:

Using AWS CLI, you can automate the creation of a CodeBuild project.

```
aws codebuild create-project --name MyBuildProject \
  --source type=GITHUB,location=https://github.com/myuser/myrepo \
  --artifacts type=NO_ARTIFACTS \
  --environment
type=LINUX_CONTAINER,environmentVariableOverrides=[{name=ENV_VAR_
NAME,value=VALUE}] \
  --service-role arn:aws:iam::ACCOUNT_ID:role/CodeBuildRole
```

This creates a build project that uses a GitHub repository as a source and doesn't store build artifacts.

---

3. Configuration with CloudFormation:

You can define AWS CodeBuild in a CloudFormation template as a resource.

yaml

```
Resources:
  MyBuildProject:
    Type: AWS::CodeBuild::Project
    Properties:
      Name: MyBuildProject
      Source:
        Type: GITHUB
        Location: https://github.com/myuser/myrepo
      Environment:
        ComputeType: BUILD_GENERAL1_SMALL
        Image: aws/codebuild/standard:4.0
        Type: LINUX_CONTAINER
      ServiceRole: arn:aws:iam::ACCOUNT_ID:role/CodeBuildRole
```

This configuration creates the same CodeBuild project using a YAML CloudFormation template.

---

4. Configuration with Terraform:

Here's how to define a CodeBuild project using Terraform:

hcl

```
resource "aws_codebuild_project" "my_build_project" {
  name = "MyBuildProject"

  source {
    type     = "GITHUB"
    location = "https://github.com/myuser/myrepo"
  }

  environment {
    compute_type = "BUILD_GENERAL1_SMALL"
```

```
  image        = "aws/codebuild/standard:4.0"
  type         = "LINUX_CONTAINER"
 }

 service_role = "arn:aws:iam::ACCOUNT_ID:role/CodeBuildRole"
}
```

This creates a CodeBuild project with Terraform using a GitHub repository as the source.

---

5. Configuration with Pulumi:

Pulumi can also manage AWS CodeBuild through its infrastructure-as-code framework. Here's an example in JavaScript:

javascript

```
const aws = require("@pulumi/aws");

const project = new aws.codebuild.Project("MyBuildProject", {
   source: {
      type: "GITHUB",
      location: "https://github.com/myuser/myrepo",
   },
   environment: {
      computeType: "BUILD_GENERAL1_SMALL",
      image: "aws/codebuild/standard:4.0",
      type: "LINUX_CONTAINER",
   },
   serviceRole: "arn:aws:iam::ACCOUNT_ID:role/CodeBuildRole",
});
```

---

3. CodeDeploy: Automate code deployments.

Automating code deployments is essential for continuous delivery in modern DevOps practices. AWS provides multiple tools and methods for automating deployments, including AWS UI, CLI, CloudFormation, Terraform, and Pulumi.

1. AWS UI

Step 1: Create an AWS CodeDeploy Application

- Sign in to the AWS Management Console.
- Navigate to **AWS CodeDeploy** and click **Create application**.
- Enter an application name.
- Choose a **Compute platform**:
    - **EC2/On-Premises** (for deploying to EC2 instances or on-prem servers)
    - **AWS Lambda** (for deploying function versions)
    - **Amazon ECS** (for blue/green deployments with ECS)

Step 2: Create a Deployment Group

- Inside your application, go to **Deployment groups** and click **Create deployment group**.
- Enter a deployment group name and choose a **Service role** with necessary permissions.
- For **EC2/On-Premises**, select target instances using Auto Scaling groups or tags.
- Configure optional **Load balancing** with an ALB if needed.
- Choose a **Deployment configuration** (e.g., OneAtATime, HalfAtATime, AllAtOnce).
- Enable **Amazon CloudWatch alarms** for monitoring.
- Click **Create deployment group**.

Step 3: Create a Deployment

- Inside the deployment group, click **Create deployment**.
- Choose the **Revision type**:
    - **Amazon S3** (for applications stored in an S3 bucket)
    - **GitHub** (for repositories)
    - **Bitbucket** (for Bitbucket repositories)
- Specify the **AppSpec file** (appspec.yml) for deployment instructions.
- Configure rollback settings if necessary.
- Click **Create deployment**.

Step 4: Monitor and Manage Deployment

- Track deployment progress in the **AWS CodeDeploy Dashboard**.
- Check logs in **AWS CloudWatch**.
- Set up automatic rollbacks in case of failures.

---

2. AWS CLI

Step 1: Install and Configure AWS CLI

Ensure the AWS CLI is installed and configured with the necessary credentials:

aws configure

Enter:

- **AWS Access Key**
- **AWS Secret Key**
- **Default region**
- **Output format** (json, yaml, text, etc.)

Step 2: Create a CodeDeploy Application

```
aws deploy create-application --application-name MyApp
```

Step 3: Create a Deployment Group

```
aws deploy create-deployment-group \
    --application-name MyApp \
    --deployment-group-name MyDeploymentGroup \
    --service-role-arn arn:aws:iam::ACCOUNT_ID:role/CodeDeployRole \
    --ec2-tag-filters Key=Name,Value=MyEC2Instance,Type=KEY_AND_VALUE
```

Replace **ACCOUNT_ID** and **MyEC2Instance** with actual values.

Step 4: Prepare the App Revision

Create an appspec.yml file:

```
yaml
version: 0.0
os: linux
files:
  - source: /
    destination: /var/www/html
hooks:
  AfterInstall:
    - location: scripts/restart_server.sh
```

```
    timeout: 300
    runas: root
```

Package and upload to S3:

```
zip -r myapp.zip *
aws s3 cp myapp.zip s3://my-deploy-bucket/
```

Step 5: Create a Deployment

```
aws deploy create-deployment \
  --application-name MyApp \
  --deployment-group-name MyDeploymentGroup \
  --s3-location bucket=my-deploy-bucket,key=myapp.zip,bundleType=zip
```

Step 6: Monitor Deployment

```
aws deploy get-deployment --deployment-id <deployment-id>
aws deploy list-deployments --application-name MyApp
```

Step 7: Rollback Deployment (If Needed)

```
aws deploy create-deployment \
  --application-name MyApp \
  --deployment-group-name MyDeploymentGroup \
```

--revision revisionType=S3,bucket=my-deploy-bucket,key=previous_version.zip,bundleType=zip

Step 8: Delete CodeDeploy Resources (If Needed)

aws deploy delete-deployment-group --application-name MyApp --deployment-group-name MyDeploymentGroup

aws deploy delete-application --application-name MyApp

---

3. AWS CloudFormation

AWS CloudFormation enables Infrastructure as Code for managing CodeDeploy. Example template:

yaml

```
Resources:
  MyCodeDeployApplication:
    Type: AWS::CodeDeploy::Application
    Properties:
      ComputePlatform: Server

  MyCodeDeployDeploymentGroup:
    Type: AWS::CodeDeploy::DeploymentGroup
    Properties:
```

```
    ApplicationName: !Ref MyCodeDeployApplication
    DeploymentConfigName: CodeDeployDefault.OneAtATime
    ServiceRoleArn: arn:aws:iam::123456789012:role/CodeDeployRole
    AutoRollbackConfiguration:
      Enabled: true
```

---

4. Terraform

Terraform allows defining CodeDeploy resources declaratively:

hcl

```
resource "aws_codedeploy_application" "my_app" {
  name = "MyApp"
  compute_platform = "Server"
}

resource "aws_codedeploy_deployment_group" "my_deployment_group" {
  application_name = aws_codedeploy_application.my_app.name
  deployment_group_name = "MyDeploymentGroup"
  service_role_arn = "arn:aws:iam::123456789012:role/CodeDeployRole"
  deployment_config_name = "CodeDeployDefault.OneAtATime"
}
```

---

5. Pulumi

Pulumi enables defining AWS CodeDeploy resources using modern programming languages. Example in TypeScript:

typescript

```
import * as aws from "@pulumi/aws";


const app = new aws.codedeploy.Application("myApp", {
    computePlatform: "Server",
});


const deploymentGroup = new aws.codedeploy.DeploymentGroup("myDeploymentGroup", {
    application: app.name,
    deploymentConfigName: "CodeDeployDefault.OneAtATime",
    serviceRole: "arn:aws:iam::123456789012:role/CodeDeployRole",
});
```

---

4. CodePipeline: Continuous integration and delivery (CI/CD).

Step 1: Create a New Pipeline

1. Sign in to the **AWS Management Console**.
2. Navigate to **AWS CodePipeline**.
3. Click **Create pipeline**.
4. Enter a **Pipeline name**.
5. Select a **Pipeline service role**:
    - Choose **New service role** (AWS creates a role for you).
    - Or select an **existing IAM role**.
6. Click **Next**.

Step 2: Add a Source Stage

1. In the **Source** section, choose a **source provider**:
    - **GitHub (V2)** → Connect your GitHub repo and select a branch.
    - **AWS CodeCommit** → Choose an existing repository.
    - **Amazon S3** → Provide the S3 bucket and object key (for zipped source files).
2. Select **Detect changes automatically** to trigger the pipeline on commits.
3. Click **Next**.

Step 3: Add a Build Stage (Optional)

1. Choose a **build provider**:
    - **AWS CodeBuild** → Create a build project or use an existing one.
    - **Jenkins** → Connect your Jenkins instance.
    - **Skip build stage** if no build is needed.
2. Configure build settings (for CodeBuild):
    - Choose a **build environment** (Ubuntu, Windows, etc.).

    - Select a **buildspec.yml** file (or enter build commands manually).
3. Click **Next**.

Step 4: Add a Deploy Stage

1. Choose a **deployment provider**:
    - **AWS CodeDeploy** → Select an application and deployment group.
    - **Amazon ECS** → Deploy to an ECS service.
    - **AWS Lambda** → Deploy a new Lambda function version.
    - **CloudFormation** → Deploy using infrastructure as code.
2. Configure deployment details.
3. Click **Next**.

Step 5: Review and Create Pipeline

1. Review the pipeline configuration.
2. Click **Create pipeline**.
3. The pipeline will **automatically start** and deploy code upon changes.

Step 6: Monitor and Manage the Pipeline

- View execution history in the **AWS CodePipeline dashboard**.
- Use **CloudWatch Logs** for debugging.
- Manually release changes if needed.

Next Steps

- **Integrate with AWS CodeBuild** for automated testing.
- **Use AWS CodeDeploy** for blue/green deployments.
- **Automate pipeline creation using Terraform or AWS CLI**.

---

2. AWS CLI Configuration (CodePipeline)

To configure CodePipeline via the AWS CLI:

aws codepipeline create-pipeline --cli-input-json file://pipeline.json

Where pipeline.json includes the configuration for source, build, and deploy stages.

CloudFormation Configuration (CodePipeline)

CloudFormation allows you to define CodePipeline resources in a template. Here's a sample configuration:

yaml

```
Resources:
  MyPipeline:
    Type: AWS::CodePipeline::Pipeline
    Properties:
      Name: MyPipeline
      RoleArn: arn:aws:iam::account-id:role/service-role/my-codepipeline-role
      ArtifactStore:
        Type: S3
        Location: my-artifact-store
      Stages:
        - Name: Source
          Actions:
            - Name: SourceAction
              ActionTypeId:
                Category: Source
                Owner: AWS
```

```
        Provider: CodeCommit
        Version: '1'
      OutputArtifacts:
        - Name: SourceOutput
      Configuration:
        RepositoryName: my-repo
        BranchName: main
  - Name: Build
    Actions:
      - Name: BuildAction
        ActionTypeId:
          Category: Build
          Owner: AWS
          Provider: CodeBuild
          Version: '1'
        InputArtifacts:
          - Name: SourceOutput
        OutputArtifacts:
          - Name: BuildOutput
        Configuration:
          ProjectName: my-codebuild-project
```

---

3. Terraform Configuration (CodePipeline)

With Terraform, define your pipeline in a .tf file:

hcl

```
resource "aws_codepipeline" "example" {
  name     = "my-pipeline"
  role_arn = "arn:aws:iam::account-id:role/service-role/my-codepipeline-role"

  artifact_store {
```

```
  location = "my-artifact-store"
  type     = "S3"
}

stage {
  name = "Source"
  action {
    name             = "SourceAction"
    category         = "Source"
    owner            = "AWS"
    provider         = "CodeCommit"
    version          = "1"
    output_artifacts = ["SourceOutput"]
    configuration    = {
      RepositoryName = "my-repo"
      BranchName     = "main"
    }
  }
}

stage {
  name = "Build"
  action {
    name             = "BuildAction"
    category         = "Build"
    owner            = "AWS"
    provider         = "CodeBuild"
    version          = "1"
    input_artifacts  = ["SourceOutput"]
    output_artifacts = ["BuildOutput"]
    configuration    = {
      ProjectName = "my-codebuild-project"
    }
  }
}
```

```
}
```

---

4. Pulumi Configuration (CodePipeline)

In Pulumi, you can define AWS CodePipeline using TypeScript or Python. Here's a TypeScript example:

typescript

```
import * as aws from "@pulumi/aws";

const pipeline = new aws.codepipeline.Pipeline("my-pipeline", {
    roleArn: "arn:aws:iam::account-id:role/service-role/my-codepipeline-role",
    artifactStore: {
        location: "my-artifact-store",
        type: "S3",
    },
    stages: [{
        name: "Source",
        actions: [{
            name: "SourceAction",
            category: "Source",
            owner: "AWS",
            provider: "CodeCommit",
            version: "1",
            outputArtifacts: ["SourceOutput"],
            configuration: {
                RepositoryName: "my-repo",
                BranchName: "main",
            },
        }],
    }, {
        name: "Build",
```

```
      actions: [{
        name: "BuildAction",
        category: "Build",
        owner: "AWS",
        provider: "CodeBuild",
        version: "1",
        inputArtifacts: ["SourceOutput"],
        outputArtifacts: ["BuildOutput"],
        configuration: {
          ProjectName: "my-codebuild-project",
        },
      }],
    }],
});
```

CodePipeline Overview for CI/CD

CodePipeline automates your software release process. It integrates with services like CodeCommit, CodeBuild, CodeDeploy, and Lambda for a complete continuous integration and delivery (CI/CD) pipeline. It enables faster and more reliable software releases by automatically building, testing, and deploying code changes.

---

5. Cloud9: Online IDE.

AWS Cloud9: A Unified Cloud IDE

AWS Cloud9 is an integrated development environment (IDE) hosted in the cloud, enabling developers to write, run, and debug code with just a browser. It offers a seamless experience for building applications directly within the AWS ecosystem, with full access to AWS resources and configurations. You can configure AWS services using different approaches:

---

1. Configure Cloud9 with AWS UI

Step 1: Create a Cloud9 Environment via AWS UI

1. Go to the AWS Cloud9 Console.
2. Click on **Create Environment**.
3. Provide a **name** for the environment (e.g., MyCloud9Environment) and an optional **description**.
4. Under **Environment Settings**, select:
    - **Environment Type**: Choose **Create a new EC2 instance for environment**.
    - **Instance Type**: Select an instance type, such as t2.micro (Free Tier eligible).
    - **Platform**: Choose **Amazon Linux 2** or **Ubuntu**.
    - **SSH or SSM**: Choose your connection type (SSH is default, or choose SSM for better security).
    - Set **Automatic Stop** to a preferred time to automatically stop after inactivity (e.g., 30 minutes).
5. Click **Next Step** and **Create Environment** to provision the Cloud9 instance.

---

2. Configure Cloud9 with AWS CLI

Step 1: Install AWS CLI in Cloud9

Open the **Cloud9 terminal** and check if AWS CLI is already installed:
aws --version
If it's not installed, install it:
sudo yum install aws-cli -y # For Amazon Linux 2

Step 2: Configure AWS CLI

aws configure

Enter your **AWS Access Key**, **Secret Access Key**, **Region**, and **Output format**.

Step 3: Create a Cloud9 Environment via CLI

To create a Cloud9 environment using the AWS CLI:

```
aws cloud9 create-environment-ec2 \
  --name MyCloud9Environment \
  --instance-type t2.micro \
  --automatic-stop-time-minutes 30 \
  --owner-arn arn:aws:iam::123456789012:user/MyUser \
  --connection-type CONNECT_SSH
```

Check the environment's status:

```
aws cloud9 describe-environments --environment-ids <environment-id>
```

---

3. Configure Cloud9 with CloudFormation

Step 1: Create a CloudFormation Template for Cloud9

Here's an example CloudFormation YAML template that creates an EC2-based Cloud9 environment:

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  Cloud9Environment:
    Type: "AWS::Cloud9::EnvironmentEC2"
    Properties:
      Name: "MyCloud9Environment"
      InstanceType: "t2.micro"  # You can choose a different instance type
```

```
    AutomaticStopTimeMinutes: 30  # Automatically stop after 30 minutes of
inactivity
    OwnerArn: !Sub arn:aws:iam::${AWS::AccountId}:root
    Description: "My Cloud9 Development Environment"
    ConnectionType: "CONNECT_SSH"  # Other options: CONNECT_SSM for
SSM-based access

Outputs:
  Cloud9EnvironmentURL:
    Description: "URL to access Cloud9 Environment"
    Value: !Sub
"https://console.aws.amazon.com/cloud9/home?region=${AWS::Region}#IDE:env
=${Cloud9Environment}"
```

Step 2: Deploy the CloudFormation Stack

- **Using AWS Console**: Upload the YAML file and create the stack.

---

2. Using AWS CLI:

```
aws cloudformation create-stack --stack-name Cloud9EnvironmentStack
--template-body file://cloud9-environment.yaml
```

Step 3: Access the Cloud9 Environment

Once the stack is created, you'll see a **Cloud9EnvironmentURL** output, which you can use to access the environment.

---

3. Configure Cloud9 with Terraform

Step 1: Install Terraform in Cloud9

Download and install Terraform in Cloud9:

```
wget
https://releases.hashicorp.com/terraform/1.5.5/terraform_1.5.5_linux_amd64.zip
unzip terraform_1.5.5_linux_amd64.zip
sudo mv terraform /usr/local/bin/
terraform --version
```

Step 2: Write Terraform Configuration for Cloud9

Create a Terraform configuration file (e.g., cloud9.tf):

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_cloud9_environment_ec2" "example" {
  name                = "MyCloud9Environment"
  instance_type       = "t2.micro"
  automatic_stop_time_minutes = 30
  connection_type     = "CONNECT_SSH"
}

output "cloud9_url" {
  value = aws_cloud9_environment_ec2.example.id
}
```

Step 3: Deploy with Terraform

Initialize Terraform:

```
terraform init
```

Apply the configuration:
terraform apply

Check the output for the **Cloud9 environment URL** and access it.

---

5. Configure Cloud9 with Pulumi

Step 1: Install Pulumi in Cloud9

Install Pulumi:
curl -fsSL https://get.pulumi.com | sh

Verify the installation:
pulumi version

Step 2: Write Pulumi Configuration for Cloud9

Create a __main__.py file for Pulumi to create a Cloud9 environment using Python:

python

```
import pulumi
import pulumi_aws as aws

cloud9_env = aws.cloud9.EnvironmentEC2("MyCloud9Environment",
    instance_type="t2.micro",
    automatic_stop_time_minutes=30,
    connection_type="CONNECT_SSH")

pulumi.export("cloud9_environment_url", cloud9_env.id)
```

Step 3: Deploy with Pulumi

Initialize a new Pulumi project:
pulumi new aws-python

Run the following to create the environment:
pulumi up

---

6. X-Ray: Debugging and tracing applications.

AWS X-Ray: Debugging and Tracing Applications

AWS X-Ray is a service that helps developers analyze and debug production applications, especially those built using microservices architecture. It provides detailed insights into the performance of your application and its components by tracing requests as they travel through various AWS services. With X-Ray, you can identify bottlenecks, errors, and latency issues in your applications.

Configuration Methods:

1. **AWS UI (Console)**:
    - Navigate to the AWS X-Ray Console.
    - Select **Get Started** or **Create a Trace**.
    - Enable tracing for your application (for example, AWS Lambda, EC2, or ECS).
    - Use the console to visualize and analyze the traces of requests processed by your application.

---

2. **AWS CLI**:

- **Install the AWS CLI** if not already installed.

Use the following command to enable X-Ray for your Lambda function:
aws lambda update-function-configuration --function-name <function-name> --tracing-config Mode=Active

To create or manage resources like sampling rules, use the aws xray command:
aws xray put-sampling-rule --sampling-rule-document file://sampling-rule.json

---

3. **AWS CloudFormation**:

Use CloudFormation to provision and configure AWS resources with X-Ray integration. For example:
yaml
```
Resources:
  MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: MyLambda
      Role: arn:aws:iam::123456789012:role/execution-role
      TracingConfig:
        Mode: Active
```

---

4. **Terraform**:

You can configure X-Ray tracing in Terraform for services like Lambda:
hcl

```
resource "aws_lambda_function" "my_lambda" {
  function_name = "my_lambda_function"
  role          = aws_iam_role.lambda_exec.arn
  tracing_config {
    mode = "Active"
  }
}
```

---

5. **Pulumi**:

Pulumi provides a declarative infrastructure setup. For example:
typescript

```
import * as aws from "@pulumi/aws";

const lambda = new aws.lambda.Function("myLambda", {
    role: myLambdaRole.arn,
    tracingConfig: {
        mode: "Active",
    },
});
```

Security, Identity, & Compliance

1.IAM (Identity and Access Management): Manage user permissions.

IAM allows you to securely control access to AWS services and resources for your users and applications. It enables you to manage who can access your resources (users) and what actions they can perform (permissions).

---

1. Configuration using AWS UI

1. **Sign in to the AWS Management Console**.
2. **Navigate to IAM**: In the Services search bar, type "IAM" and select it.
3. **Create User**: Go to the "Users" section and click "Add user." You can assign programmatic or console access.
4. **Set Permissions**: Select permissions for the user. You can attach existing policies or create a custom one.
5. **Review and Create**: Review the settings and create the user. You'll get access keys to interact with AWS via CLI or SDK.

---

2. Configuration using AWS CLI

1. **Install AWS CLI** and configure it using the aws configure command.

Create IAM User:
aws iam create-user --user-name my-user

Attach a Policy:
aws iam attach-user-policy --user-name my-user --policy-arn arn:aws:iam::aws:policy/AdministratorAccess

Create Access Keys:
aws iam create-access-key --user-name my-user

---

3. Configuration using AWS CloudFormation

CloudFormation Template:
yaml

```
Resources:
  MyUser:
    Type: AWS::IAM::User
    Properties:
      UserName: my-user
      Policies:
        - PolicyName: MyUserPolicy
          PolicyDocument:
            Version: "2012-10-17"
            Statement:
              - Effect: Allow
                Action: ["s3:ListBucket"]
                Resource: "arn:aws:s3:::my-bucket"
```

Create Stack: Use the CloudFormation Console or CLI to create the stack.
aws cloudformation create-stack --stack-name MyStack --template-body file://template.yaml

---

4. Configuration using Terraform

Terraform Configuration:
hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_iam_user" "my_user" {
  name = "my-user"
}

resource "aws_iam_user_policy" "my_user_policy" {
  name   = "MyUserPolicy"
  user   = aws_iam_user.my_user.name
  policy = jsonencode({
    Version = "2012-10-17"
    Statement = [
      {
        Action   = "s3:ListBucket"
        Effect   = "Allow"
        Resource = "arn:aws:s3:::my-bucket"
      }
    ]
  })
}
```

Apply the Configuration:
terraform init
terraform apply

---

5. Configuration using Pulumi

Pulumi Configuration (using TypeScript):
typescript

```
import * as aws from "@pulumi/aws";

const user = new aws.iam.User("my-user");

const policy = new aws.iam.UserPolicy("my-user-policy", {
    user: user.name,
    policy: JSON.stringify({
        Version: "2012-10-17",
        Statement: [
            {
                Action: "s3:ListBucket",
                Effect: "Allow",
                Resource: "arn:aws:s3:::my-bucket"
            }
        ]
    })
});
```

Run Pulumi:
pulumi up

---

2. Cognito: User authentication and identity.

AWS Cognito Overview

AWS **Cognito** is a fully managed service that provides user authentication, authorization, and management for your web and mobile apps. It allows you to

create and manage user pools for authentication, integrate third-party identity providers, and securely handle user sign-ins.

---

1. Configuration with AWS UI (Management Console)

1. **Navigate to Cognito:**
    - Go to the AWS Management Console.
    - Search for and select **Amazon Cognito**.
2. **Create a User Pool:**
    - Click **Manage User Pools** > **Create a Pool**.
    - Provide a pool name, configure settings like sign-in options (email, username), multi-factor authentication (MFA), etc.
    - Define app clients, such as a web or mobile app, and save the pool settings.
3. **Integrate with Identity Providers:**
    - You can integrate with external identity providers like Facebook, Google, or SAML.
    - Configure this under **Federation** in the user pool settings.
4. **Setup Lambda Triggers (Optional):**
    - Add Lambda triggers for customized authentication flows like pre-sign-up, post-sign-in, etc.

---

2. Configuration with AWS CLI

1. **Install AWS CLI:**
    - Ensure AWS CLI is installed (aws --version).

Create a User Pool:
aws cognito-idp create-user-pool --pool-name MyUserPool

Create an App Client:
aws cognito-idp create-user-pool-client --user-pool-id <PoolId> --client-name MyAppClient

2. **Configure MFA and other settings** through CLI commands for user pools, groups, etc.

---

3. Configuration with CloudFormation

User Pool Resource in CloudFormation: Define the Cognito user pool in your CloudFormation template.
yaml

```
Resources:
  MyCognitoUserPool:
    Type: AWS::Cognito::UserPool
    Properties:
      UserPoolName: MyUserPool
      AliasAttributes:
        - email
      AutoVerifiedAttributes:
        - email
```

User Pool Client:
yaml

```
MyCognitoUserPoolClient:
  Type: AWS::Cognito::UserPoolClient
  Properties:
    ClientName: MyAppClient
    UserPoolId: !Ref MyCognitoUserPool
```

GenerateSecret: false

---

4. Configuration with Terraform

Terraform User Pool and App Client:
hcl

```
resource "aws_cognito_user_pool" "my_user_pool" {
  name = "my_user_pool"
  alias_attributes = ["email"]
  auto_verified_attributes = ["email"]
}

resource "aws_cognito_user_pool_client" "my_user_pool_client" {
  name         = "my_app_client"
  user_pool_id = aws_cognito_user_pool.my_user_pool.id
  generate_secret = false
}
```

1. **Terraform Apply:** Run terraform init and terraform apply to create the resources.

---

5. Configuration with Pulumi

Pulumi User Pool and Client (using TypeScript):
typescript

```
import * as aws from "@pulumi/aws";

const userPool = new aws.cognito.UserPool("myUserPool", {
```

```
  aliasAttributes: ["email"],
  autoVerifiedAttributes: ["email"],
});

const userPoolClient = new aws.cognito.UserPoolClient("myAppClient", {
  userPoolId: userPool.id,
  generateSecret: false,
});
```

- **Run Pulumi:**
    - Run pulumi up to deploy the resources.

---

3. Secrets Manager: Securely manage secrets.

AWS Secrets Manager: Securely Manage Secrets

AWS Secrets Manager is a service that helps you securely store, manage, and retrieve sensitive information such as API keys, passwords, database credentials, and other secrets. It offers the ability to rotate, manage access policies, and integrate seamlessly with AWS services.

---

1. AWS Secrets Manager via AWS UI (Console):

1. Open the **AWS Management Console**.
2. Navigate to **Secrets Manager** under the **Security, Identity & Compliance** section.
3. Click **Store a new secret**.
4. Choose the type of secret (e.g., Other type of secrets, RDS, etc.).
5. Enter the secret key-value pairs, configure any additional options (rotation, tags, etc.), and click **Next**.
6. Define a **name** for your secret and set permissions for access.
7. Review and create the secret.

---

2. AWS Secrets Manager via AWS CLI:

To create a secret with the AWS CLI, you can use the create-secret command:

```
aws secretsmanager create-secret \
   --name MySecretName \
   --description "My secret description" \
   --secret-string '{"username":"myuser","password":"mypassword"}'
```

You can use get-secret-value to retrieve the secret:

```
aws secretsmanager get-secret-value --secret-id MySecretName
```

---

3. AWS Secrets Manager via CloudFormation:

To configure a secret using AWS CloudFormation, you define the AWS::SecretsManager::Secret resource in a CloudFormation template:

yaml

```
Resources:
  MySecret:
    Type: AWS::SecretsManager::Secret
    Properties:
      Name: MySecretName
      Description: "My secret description"
      SecretString: '{"username":"myuser","password":"mypassword"}'
```

You can deploy this with aws cloudformation create-stack.

---

4. AWS Secrets Manager via Terraform:

With Terraform, you can create a secret like this:

hcl

```
resource "aws_secretsmanager_secret" "example" {
  name        = "MySecretName"
  description = "My secret description"
}

resource "aws_secretsmanager_secret_version" "example" {
  secret_id     = aws_secretsmanager_secret.example.id
  secret_string = "{\"username\":\"myuser\",\"password\":\"mypassword\"}"
}
```

Run terraform apply to create the secret.

---

5. AWS Secrets Manager via Pulumi:

In Pulumi (using TypeScript as an example):

typescript

```
import * as aws from "@pulumi/aws";

const secret = new aws.secretsmanager.Secret("mySecret", {
    description: "My secret description",
});

const secretVersion = new aws.secretsmanager.SecretVersion("mySecretVersion", {
    secretId: secret.id,
    secretString: "{\"username\":\"myuser\",\"password\":\"mypassword\"}",
});
```

Run pulumi up to deploy this secret configuration.

---

4. Certificate Manager: Manage SSL/TLS certificates.

AWS Certificate Manager (ACM) Overview

AWS Certificate Manager (ACM) helps you easily provision, manage, and deploy SSL/TLS certificates for your AWS-based applications. ACM simplifies the process of securing websites, applications, and APIs by automating certificate issuance, renewal, and deployment.

1. AWS Certificate Manager with AWS Management Console (UI)

1. **Sign in to AWS Management Console**.
2. **Navigate to ACM**: From the AWS Management Console, go to **Services** > **Certificate Manager**.
3. **Request a certificate**: Click on "Request a certificate" and choose either a public or private certificate.
4. **Validation**: Depending on your choice, validate the certificate either via DNS or email validation.
5. **Deploy**: Once validated, you can deploy the certificate to your services like Elastic Load Balancer (ELB), CloudFront, or API Gateway.

---

2. AWS Certificate Manager with AWS CLI

Use the following commands to interact with ACM from the AWS CLI:

Request a public certificate:
aws acm request-certificate --domain-name example.com --validation-method DNS

List certificates:
aws acm list-certificates

Delete a certificate:
aws acm delete-certificate --certificate-arn arn:aws:acm:region:account-id:certificate/certificate-id

Describe certificate:
aws acm describe-certificate --certificate-arn arn:aws:acm:region:account-id:certificate/certificate-id

---

3. AWS Certificate Manager with CloudFormation

You can use **CloudFormation** to automate the management of certificates in ACM.

Example YAML configuration:

yaml

```
Resources:
  MySSLCertificate:
    Type: AWS::ACM::Certificate
    Properties:
```

```
    DomainName: example.com
    ValidationMethod: DNS
    SubjectAlternativeNames:
      - www.example.com
```

Deploy this template using the AWS CLI:

aws cloudformation create-stack --stack-name MyACMStack --template-body file://certificate-template.yaml

---

4. AWS Certificate Manager with Terraform

In **Terraform**, you can manage ACM certificates with the aws_acm_certificate resource.

Example configuration:

hcl

```
resource "aws_acm_certificate" "example" {
  domain_name       = "example.com"
  validation_method = "DNS"
  subject_alternative_names = ["www.example.com"]
}

resource "aws_route53_record" "example_com_validation" {
  zone_id = "<Route53 Zone ID>"
  name    = "_<validation_token>.<domain_name>"
  type    = "CNAME"
  ttl     = 60
  records = ["<validation_value>"]
}
```

Run the Terraform commands:

terraform init
terraform apply

---

5. AWS Certificate Manager with Pulumi

Pulumi allows you to manage AWS resources using your preferred programming language.

Example Pulumi configuration in **JavaScript**:

javascript

```
const aws = require("@pulumi/aws");

const certificate = new aws.acm.Certificate("exampleCertificate", {
    domainName: "example.com",
    validationMethod: "DNS",
    subjectAlternativeNames: ["www.example.com"],
});

exports.certificateArn = certificate.arn;
```

Deploy the configuration using the following:

pulumi up

---

5. GuardDuty: Threat detection.

AWS GuardDuty Overview:

AWS GuardDuty is a threat detection service that continuously monitors for malicious activity and unauthorized behavior to protect your AWS accounts and workloads. It analyzes various AWS data sources such as VPC flow logs, AWS CloudTrail event logs, and DNS logs to identify potential threats.

---

1. AWS Management Console (UI):

- **Step 1**: Sign in to the AWS Management Console.
- **Step 2**: Navigate to **GuardDuty** under the Security, Identity, and Compliance section.
- **Step 3**: Click **Get Started** to enable GuardDuty in your desired region.
- **Step 4**: Configure additional settings (e.g., Data Sources, Trusted IP Lists, etc.), and click **Enable GuardDuty**.

---

2. AWS CLI:

You can enable GuardDuty using the AWS CLI with the following command:

aws guardduty create-detector --enable

To disable:

aws guardduty delete-detector --detector-id <detector-id>

---

3. AWS CloudFormation:

To create and configure GuardDuty using AWS CloudFormation, you can define it in your CloudFormation template:

yaml

```
Resources:
  GuardDutyDetector:
    Type: AWS::GuardDuty::Detector
    Properties:
      Enable: true
```

Then deploy the CloudFormation stack using:

```
aws cloudformation create-stack --stack-name guardduty-stack --template-body file://template.yaml
```

---

4. Terraform:

To configure GuardDuty with Terraform:

hcl

```
resource "aws_guardduty_detector" "example" {
  enable = true
}
```

Then apply the configuration:

```
terraform apply
```

---

5. Pulumi:

To configure GuardDuty with Pulumi (in JavaScript):

javascript

```
const aws = require("@pulumi/aws");

const detector = new aws.guardduty.Detector("example", {
    enable: true,
});
```

Run the Pulumi update command:

```
pulumi up
```

---

6. Inspector: Security assessment.

Amazon Inspector: Security Assessment

Amazon Inspector is an automated security assessment service that helps identify vulnerabilities in EC2 instances, container images, and Lambda functions. It provides actionable insights into security risks, compliance violations, and best practices to enhance your AWS security posture.

Configuration Methods

1. AWS Management Console (UI)

1. Sign in to the AWS Console.
2. Navigate to **Amazon Inspector**.
3. Enable Amazon Inspector for your AWS environment.
4. Set up an **assessment target** by selecting EC2 instances or container images.
5. Create an **assessment template** (e.g., network reachability, security best practices).

6. Run the assessment and review the findings in the console.

---

2. AWS CLI

Use the AWS CLI to automate security assessments.

Start an assessment:

sh

```
aws inspector2 start-assessment-run --assessment-target-arn <target-arn>
--assessment-template-arn <template-arn>
```

List findings:

sh

```
aws inspector2 list-findings --assessment-run-arn <run-arn>
```

---

3. AWS CloudFormation

Define security assessment configurations as code using AWS CloudFormation.

Example CloudFormation Template (YAML):

yaml

```
Resources:
  InspectorAssessmentTarget:
    Type: AWS::Inspector2::AssessmentTarget
    Properties:
      Name: "MyAssessmentTarget"
      ResourceGroupArn:
arn:aws:resource-groups:region:account-id:group/my-resource-group
```

```
  InspectorAssessmentTemplate:
    Type: AWS::Inspector2::AssessmentTemplate
    Properties:
      AssessmentTargetArn: !Ref InspectorAssessmentTarget
      DurationInSeconds: 3600
      RulesPackageArns:
        - arn:aws:inspector2:rules-package-arn
```

---

4. Terraform

Use Terraform to configure Amazon Inspector.

Example Terraform Configuration:

```
resource "aws_inspector_assessment_target" "example" {
  name = "MyAssessmentTarget"
  resource_group_arn =
"arn:aws:resource-groups:region:account-id:group/my-resource-group"
}

resource "aws_inspector_assessment_template" "example" {
  name = "MyAssessmentTemplate"
  assessment_target_arn = aws_inspector_assessment_target.example.arn
  duration_in_seconds = 3600
  rules_package_arns = ["arn:aws:inspector2:rules-package-arn"]
}
```

---

5. Pulumi

Use Pulumi with TypeScript, Python, or Go to configure Amazon Inspector.

Example Pulumi Configuration (TypeScript):

typescript

```
import * as aws from "@pulumi/aws";

// Define the Assessment Target
const assessmentTarget = new
aws.inspector2.AssessmentTarget("myAssessmentTarget", {
   resourceGroupArn:
"arn:aws:resource-groups:region:account-id:group/my-resource-group",
});

// Define the Assessment Template
const assessmentTemplate = new
aws.inspector2.AssessmentTemplate("myAssessmentTemplate", {
   assessmentTargetArn: assessmentTarget.arn,
   durationInSeconds: 3600,
   rulesPackageArns: ["arn:aws:inspector2:rules-package-arn"],
});
```

---

7. Macie: Data security and privacy.

Introduction to AWS Macie

AWS Macie is a fully managed data security and privacy service that uses machine learning to discover, classify, and protect sensitive data in AWS. It helps identify personally identifiable information (PII), financial data, and other critical content, ensuring compliance with security and privacy standards.

---

1. Configuring AWS Macie via AWS UI

1. **Sign in** to the AWS Management Console.
2. Navigate to **Amazon Macie** service.

3. Click **Enable Macie** and choose the AWS Region.
4. Configure **S3 buckets** to scan for sensitive data.
5. Set up **Findings notifications** via AWS Security Hub or Amazon EventBridge.
6. Review settings and click **Save** to activate Macie.

---

2. Configuring AWS Macie via AWS CLI

Enable Macie:

sh
aws macie2 enable-macie --region us-east-1

Create a classification job:

sh

aws macie2 create-classification-job --name "SensitiveDataScan" --region us-east-1 \
  --s3-job-definition '{"bucketDefinitions":[{"accountId":"<AWS_ACCOUNT_ID>","buckets":["<S3_BUCKET_NAME>"]}]}'

Check Macie status:

sh
aws macie2 get-macie-session

---

3. Configuring AWS Macie with AWS CloudFormation

Create a **CloudFormation template** (YAML format) to enable Macie:

yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  MacieSession:
    Type: AWS::Macie::Session
    Properties:
      Status: ENABLED
```

Deploy with AWS CLI:

sh

```
aws cloudformation create-stack --stack-name MacieStack --template-body
file://macie.yaml
```

---

4. Configuring AWS Macie with Terraform

Create a main.tf file:

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_macie2_account" "macie" {
  status = "ENABLED"
}
```

Deploy with Terraform:

sh

terraform init
terraform apply -auto-approve

---

5. Configuring AWS Macie with Pulumi (Python)

Create a **Pulumi Python script** (__main__.py):

python

```
import pulumi
import pulumi_aws as aws

macie = aws.macie2.Account("macieAccount", status="ENABLED")
pulumi.export("macie_status", macie.status)
```

Deploy with Pulumi:

sh

```
pulumi up
```

---

8. Security Hub: Centralized security view.

AWS Security Hub: Centralized Security View

AWS Security Hub is a security service that provides a comprehensive view of security alerts and compliance status across AWS accounts. It aggregates,

organizes, and prioritizes security findings from AWS services like AWS GuardDuty, AWS Inspector, AWS Macie, and third-party security tools.

Configuration Methods

1. AWS UI (Console)

- Sign in to the **AWS Management Console**
- Navigate to **Security Hub** from the **AWS Services** menu
- Click **Enable Security Hub**
- Configure **security standards** (e.g., AWS Foundational Security Best Practices, CIS AWS Benchmark)
- Enable integrations with AWS services and third-party tools

---

2. AWS CLI

Enable Security Hub using the CLI:

aws securityhub enable-security-hub

List available security standards:

aws securityhub get-enabled-standards

Enable a security standard (replace <ARN> with the actual ARN):

aws securityhub batch-enable-standards --standards-subscriptions '[{"StandardsArn": "<ARN>"}]'

---

3. AWS CloudFormation

Create a CloudFormation template (security-hub.yml):

yaml

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  SecurityHub:
    Type: "AWS::SecurityHub::Hub"
```

Deploy it:

```
aws cloudformation create-stack --stack-name SecurityHubStack --template-body file://security-hub.yml
```

---

4. Terraform

Define Security Hub in main.tf:

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_securityhub_account" "example" {}
```

Deploy using Terraform

```
terraform init
terraform apply -auto-approve
```

---

5. Pulumi (Python)

python

```
import pulumi
import pulumi_aws as aws

security_hub = aws.securityhub.Account("securityHub")
```

Deploy with Pulumi:

```
pulumi up
```

---

9. WAF (Web Application Firewall): Protect web applications.

AWS WAF is a security service that helps protect web applications from common threats like SQL injection, cross-site scripting (XSS), and other web exploits. It allows you to define rules to filter and block malicious requests before they reach your applications.

---

1. Configuring AWS WAF using AWS Management Console (UI)

1. Navigate to the **AWS WAF** service in the AWS console.
2. Click **Create Web ACL** and provide a name.
3. Choose the resource to protect (Application Load Balancer, CloudFront, API Gateway, or App Runner).
4. Add **rules** (AWS Managed Rules, rate-based rules, or custom rules).

5. Configure **rule action** (allow, block, or count).
6. Review and deploy the Web ACL.

---

2. Configuring AWS WAF using AWS CLI

Create a Web ACL using AWS CLI:

```
aws wafv2 create-web-acl --name "MyWebACL" --scope REGIONAL \
--default-action Block={} \
--visibility-config
SampledRequestsEnabled=true,CloudWatchMetricsEnabled=true,MetricName="M
yWebACLMetric" \
--region us-east-1
```

To list existing Web ACLs:

```
aws wafv2 list-web-acls --scope REGIONAL --region us-east-1
```

---

3. Configuring AWS WAF using AWS CloudFormation

Create a CloudFormation template (waf.yaml):

```
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  WebACL:
    Type: AWS::WAFv2::WebACL
    Properties:
      Name: MyWebACL
      Scope: REGIONAL
      DefaultAction:
        Block: {}
      VisibilityConfig:
```

```
      SampledRequestsEnabled: true
      CloudWatchMetricsEnabled: true
      MetricName: MyWebACLMetric
```

Deploy using AWS CLI:

```
aws cloudformation create-stack --stack-name MyWAFStack --template-body file://waf.yaml
```

---

4. Configuring AWS WAF using Terraform

Create a Terraform configuration file (waf.tf):

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_wafv2_web_acl" "my_waf_acl" {
  name  = "MyWebACL"
  scope = "REGIONAL"

  default_action {
    block {}
  }

  visibility_config {
    cloudwatch_metrics_enabled = true
    metric_name                = "MyWebACLMetric"
    sampled_requests_enabled   = true
  }
}
```

Deploy using Terraform

terraform init
terraform apply -auto-approve

---

5. Configuring AWS WAF using Pulumi (Python)

Create a Pulumi project and use the following Python code:

python

```
import pulumi
import pulumi_aws as aws

web_acl = aws.wafv2.WebAcl("myWebACL",
    scope="REGIONAL",
    default_action={"block": {}},
    visibility_config={
        "sampled_requests_enabled": True,
        "cloudwatch_metrics_enabled": True,
        "metric_name": "MyWebACLMetric",
    }
)

pulumi.export("web_acl_id", web_acl.id)
```

Deploy using Pulumi:

pulumi up

---

10. Shield: DDoS protection.

AWS WAF (Web Application Firewall)

AWS WAF is a security service that helps protect web applications from common threats like SQL injection, cross-site scripting (XSS), and other web exploits. It allows you to define rules to filter and block malicious requests before they reach your applications.

---

1. Configuring AWS WAF using AWS Management Console (UI)

1. Navigate to the **AWS WAF** service in the AWS console.
2. Click **Create Web ACL** and provide a name.
3. Choose the resource to protect (Application Load Balancer, CloudFront, API Gateway, or App Runner).
4. Add **rules** (AWS Managed Rules, rate-based rules, or custom rules).
5. Configure **rule action** (allow, block, or count).
6. Review and deploy the Web ACL.

---

2. Configuring AWS WAF using AWS CLI

Create a Web ACL using AWS CLI:

```
aws wafv2 create-web-acl --name "MyWebACL" --scope REGIONAL \
--default-action Block={} \
--visibility-config
SampledRequestsEnabled=true,CloudWatchMetricsEnabled=true,MetricName="M
yWebACLMetric" \
--region us-east-1
```

To list existing Web ACLs:

```
aws wafv2 list-web-acls --scope REGIONAL --region us-east-1
```

---

3. Configuring AWS WAF using AWS CloudFormation

Create a CloudFormation template (waf.yaml):

```
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  WebACL:
    Type: AWS::WAFv2::WebACL
    Properties:
      Name: MyWebACL
      Scope: REGIONAL
      DefaultAction:
        Block: {}
      VisibilityConfig:
        SampledRequestsEnabled: true
        CloudWatchMetricsEnabled: true
        MetricName: MyWebACLMetric
```

Deploy using AWS CLI:

```
aws cloudformation create-stack --stack-name MyWAFStack --template-body file://waf.yaml
```

---

4. Configuring AWS WAF using Terraform

Create a Terraform configuration file (waf.tf):

```
provider "aws" {
  region = "us-east-1"
}
```

```
resource "aws_wafv2_web_acl" "my_waf_acl" {
  name  = "MyWebACL"
  scope = "REGIONAL"

  default_action {
    block {}
  }

  visibility_config {
    cloudwatch_metrics_enabled = true
    metric_name                = "MyWebACLMetric"
    sampled_requests_enabled   = true
  }
}
```

Deploy using Terraform:

```
terraform init
terraform apply -auto-approve
```

---

5. Configuring AWS WAF using Pulumi (Python)

Create a Pulumi project and use the following Python code:

```
python

import pulumi
import pulumi_aws as aws

web_acl = aws.wafv2.WebAcl("myWebACL",
    scope="REGIONAL",
    default_action={"block": {}},
    visibility_config={
```

```
        "sampled_requests_enabled": True,
        "cloudwatch_metrics_enabled": True,
        "metric_name": "MyWebACLMetric",
    }
)

pulumi.export("web_acl_id", web_acl.id)
```

Deploy using Pulumi:

```
pulumi up
```

Analytics

1. Athena: Query data in S3 using SQL.

Introduction to Amazon Athena

Amazon Athena is a **serverless, interactive query service** that allows you to run **SQL queries** on data stored in **Amazon S3** without the need for complex ETL processes. It uses **Presto** and **Apache Hive** under the hood, making it efficient for analyzing structured, semi-structured, and unstructured data.

Configuration Methods

1. AWS UI (Console) Configuration

- Navigate to the **AWS Management Console** → Open **Athena**.
- Set up a **Query Result Location** in an **S3 bucket**.
- Create a new **database** and define tables using CREATE EXTERNAL TABLE SQL commands.

- Start querying S3 data using standard **SQL queries**.

---

2. AWS CLI Configuration

```
# Set the output location for query results
aws athena start-query-execution \
    --query-string "SELECT * FROM my_database.my_table LIMIT 10;" \
    --query-execution-context Database=my_database \
    --result-configuration OutputLocation=s3://my-athena-query-results/
```

---

3. AWS CloudFormation Configuration

yaml

```
Resources:
  AthenaWorkGroup:
    Type: AWS::Athena::WorkGroup
    Properties:
      Name: MyAthenaWorkGroup
      WorkGroupConfiguration:
        ResultConfiguration:
          OutputLocation: s3://my-athena-query-results/
```

---

4. Terraform Configuration

hcl

```
resource "aws_athena_database" "example" {
  name   = "my_database"
  bucket = "my-athena-query-results"
}
```

```
resource "aws_athena_workgroup" "example" {
  name = "my_workgroup"
  configuration {
    result_configuration {
      output_location = "s3://my-athena-query-results/"
    }
  }
}
```

---

5. Pulumi Configuration (Python)

python

```
import pulumi
import pulumi_aws as aws

s3_bucket = aws.s3.Bucket("athena-results")

athena_workgroup = aws.athena.Workgroup(
    "my_workgroup",
    configuration={
        "result_configuration": {
            "output_location": s3_bucket.arn
        }
    }
)

pulumi.export("athena_workgroup", athena_workgroup.name)
```

---

2. Glue: ETL (Extract, Transform, Load) service.

AWS Glue: ETL (Extract, Transform, Load) Service

AWS Glue is a serverless data integration service that allows you to discover, prepare, and transform data for analytics, machine learning, and application development. It simplifies ETL (Extract, Transform, Load) processes by automatically generating code, handling schema evolution, and integrating with various AWS services like S3, Redshift, and Athena.

AWS Glue Configuration Methods

1. AWS Management Console (UI)

1. Navigate to the **AWS Glue** service in the AWS Console.
2. Create a **Glue Data Catalog** to store metadata.
3. Define **Crawlers** to scan data sources and populate the catalog.
4. Create an **ETL Job**, select a data source (S3, RDS, etc.), and specify transformations using Glue Studio or PySpark.
5. Run and monitor the job.

---

2. AWS CLI

Install AWS CLI & Configure
aws configure

Create a Glue Job
sh

```
aws glue create-job \
   --name my-glue-job \
   --role GlueServiceRole \
   --command Name=glueetl,ScriptLocation=s3://my-bucket/scripts/etl.py
```

Start the Job
aws glue start-job-run --job-name my-glue-job

Monitor Job Status
aws glue get-job-run --job-name my-glue-job --run-id <run_id>

---

3. AWS CloudFormation

Define a CloudFormation template (glue-stack.yml) to create a Glue job.

yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  GlueJob:
    Type: AWS::Glue::Job
    Properties:
      Name: MyGlueJob
      Role: arn:aws:iam::123456789012:role/GlueServiceRole
      Command:
        Name: glueetl
        ScriptLocation: s3://my-bucket/scripts/etl.py
```

Deploy the stack:

aws cloudformation create-stack --stack-name GlueStack --template-body file://glue-stack.yml

---

4. Terraform

Create a Terraform configuration file (main.tf) for AWS Glue.

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_glue_job" "my_job" {
  name     = "my-glue-job"
  role_arn = "arn:aws:iam::123456789012:role/GlueServiceRole"

  command {
    script_location = "s3://my-bucket/scripts/etl.py"
    name            = "glueetl"
  }
}
```

Deploy with:

```
terraform init
terraform apply -auto-approve
```

---

5. Pulumi (Python)

Create a Pulumi Python script (__main__.py) to configure AWS Glue.

python

```
import pulumi
import pulumi_aws as aws
```

```
glue_job = aws.glue.Job("myGlueJob",
   role_arn="arn:aws:iam::123456789012:role/GlueServiceRole",
   command={
      "script_location": "s3://my-bucket/scripts/etl.py",
      "name": "glueetl",
   }
)

pulumi.export("glue_job_name", glue_job.name)
```

Deploy with:

```
pulumi up
```

---

3. QuickSight: Business intelligence and data visualization.

Amazon QuickSight is a fast, cloud-powered business intelligence (BI) service built for the cloud. It enables users to visualize data, perform ad-hoc analysis, and derive insights from data. With QuickSight, businesses can create interactive dashboards and share them with others for collaboration. It integrates seamlessly with various AWS services, enabling quick analysis of data stored in databases, data lakes, and more.

---

1. Configuration via AWS UI (Console):

- **Sign in to AWS Console**: Go to the Amazon QuickSight console.
- **Create a Dataset**: Choose a data source (e.g., S3, Redshift, or RDS), upload data, and define transformations.
- **Build Dashboards**: Create and customize dashboards with charts, tables, and visualizations based on the dataset.

- **Share Dashboards**: You can share the dashboards with users or groups within your AWS account.

---

2. Configuration via AWS CLI:

You can use AWS CLI commands to manage QuickSight, such as creating and managing data sources, datasets, and dashboards.

Example to list QuickSight users:

aws quicksight list-users --aws-account-id <account-id> --namespace default

You can also automate dataset creation and dashboard publishing with the CLI.

---

3. Configuration via CloudFormation:

You can define QuickSight resources in a CloudFormation template. This allows you to automate and manage QuickSight configurations in a repeatable way.

Example CloudFormation template for QuickSight:

```
yaml
Resources:
  MyQuickSightDashboard:
    Type: AWS::QuickSight::Dashboard
    Properties:
      AwsAccountId: !Ref AwsAccountId
      DashboardId: "my-dashboard"
      Name: "My Dashboard"
      SourceEntity:
        SourceTemplate:
          DataSetReferences:
```

```
      - DataSetArn: !Ref MyQuickSightDatasetArn
    Arn: arn:aws:quicksight:us-east-1:aws-account-id:template/TemplateName
```

---

4. Configuration via Terraform:

With Terraform, you can manage Amazon QuickSight resources such as datasets, dashboards, and users.

Example Terraform configuration:

```
resource "aws_quicksight_dashboard" "my_dashboard" {
  aws_account_id = "<aws-account-id>"
  dashboard_id   = "my-dashboard"
  name           = "My Dashboard"
  source_entity {
    source_template {
      data_set_references {
        data_set_arn = "<dataset-arn>"
      }
      arn = "arn:aws:quicksight:us-east-1:<account-id>:template/TemplateName"
    }
  }
}
```

---

5. Configuration via Pulumi:

With Pulumi, you can define QuickSight resources in code and manage them using the Pulumi CLI.

Example Pulumi configuration (using TypeScript):

typescript

```
import * as aws from "@pulumi/aws";

const dashboard = new aws.quicksight.Dashboard("my-dashboard", {
   awsAccountId: "<aws-account-id>",
   dashboardId: "my-dashboard",
   name: "My Dashboard",
   sourceEntity: {
      sourceTemplate: {
         dataSetReferences: [{
            dataSetArn: "<dataset-arn>"
         }],
         arn: "arn:aws:quicksight:us-east-1:<account-id>:template/TemplateName"
      }
   }
});
```

---

4. EMR (Elastic MapReduce): Big data processing.

Amazon EMR (Elastic MapReduce) - Introduction & Configuration Guide

Amazon EMR is a managed big data processing service that allows you to run large-scale distributed data processing frameworks such as **Apache Spark, Apache Hadoop, Hive, and Presto**. It simplifies running analytics, machine learning, and big data workloads without the need for on-premises infrastructure.

---

1. Configuration via AWS UI (Console)

1. **Sign in to AWS Console** → Navigate to **Amazon EMR**.
2. **Create Cluster** → Click on **Create Cluster** and configure:
    - **Cluster Name**: Set a name for your cluster.

    - **Release Version**: Choose an appropriate EMR version.
    - **Applications**: Select Hadoop, Spark, or other required frameworks.
    - **Instance Type**: Choose EC2 instance types for Master, Core, and Task nodes.
    - **Storage & Security**: Configure S3 storage, IAM roles, and security groups.
3. **Launch Cluster** → Click **Create** and wait for the cluster to start.
4. **Monitor & Run Jobs** → Use **Amazon EMR Studio** or the console to submit jobs.

---

2. Configuration via AWS CLI

You can create and manage an EMR cluster using the AWS CLI.

Create an EMR Cluster

```
aws emr create-cluster --name "MyEMRCluster" --release-label emr-6.6.0 \
  --applications Name=Hadoop Name=Spark \
  --log-uri s3://my-bucket/logs/ \
  --instance-groups
InstanceGroupType=MASTER,InstanceCount=1,InstanceType=m5.xlarge \
                InstanceGroupType=CORE,InstanceCount=2,InstanceType=m5.xlarge
\
  --use-default-roles
```

List Running Clusters

```
aws emr list-clusters --active
```

Terminate Cluster

```
aws emr terminate-clusters --cluster-ids <cluster-id>
```

---

3. Configuration via AWS CloudFormation

Define an **EMR Cluster** using **CloudFormation** for automated provisioning.

CloudFormation Template (YAML)

yaml

```
Resources:
  MyEMRCluster:
    Type: AWS::EMR::Cluster
    Properties:
      Name: "MyEMRCluster"
      ReleaseLabel: "emr-6.6.0"
      Instances:
        MasterInstanceGroup:
          InstanceType: "m5.xlarge"
          InstanceCount: 1
        CoreInstanceGroup:
          InstanceType: "m5.xlarge"
          InstanceCount: 2
      Applications:
        - Name: "Hadoop"
        - Name: "Spark"
      ServiceRole: "EMR_DefaultRole"
      JobFlowRole: "EMR_EC2_DefaultRole"
      LogUri: "s3://my-bucket/logs/"
```

Deploy this stack using the AWS CloudFormation console or CLI.

aws cloudformation create-stack --stack-name my-emr-cluster --template-body file://emr-template.yaml

---

4. Configuration via Terraform

You can use **Terraform** to deploy EMR clusters with infrastructure as code.

Terraform EMR Configuration

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_emr_cluster" "my_emr" {
  name          = "MyEMRCluster"
  release_label = "emr-6.6.0"
  applications  = ["Hadoop", "Spark"]

  ec2_attributes {
    instance_profile = "EMR_EC2_DefaultRole"
    key_name         = "my-key-pair"
  }

  master_instance_group {
    instance_type = "m5.xlarge"
    instance_count = 1
  }

  core_instance_group {
    instance_type = "m5.xlarge"
    instance_count = 2
  }

  log_uri = "s3://my-bucket/logs/"
}
```

Apply the Terraform configuration:

terraform init
terraform apply

---

5. Configuration via Pulumi

Pulumi allows you to define an EMR cluster using **TypeScript, Python, or Go**.

Pulumi EMR Configuration (TypeScript)

typescript

```
import * as aws from "@pulumi/aws";

const emrCluster = new aws.emr.Cluster("my-emr-cluster", {
    releaseLabel: "emr-6.6.0",
    applications: ["Hadoop", "Spark"],
    masterInstanceGroup: {
        instanceType: "m5.xlarge",
        instanceCount: 1,
    },
    coreInstanceGroup: {
        instanceType: "m5.xlarge",
        instanceCount: 2,
    },
    ec2Attributes: {
        instanceProfile: "EMR_EC2_DefaultRole",
    },
    logUri: "s3://my-bucket/logs/",
});
```

Deploy using Pulumi CLI:

pulumi up

---

5. Kinesis: Real-time data streaming.

Introduction to AWS Kinesis

Amazon Kinesis is a real-time data streaming service that allows you to collect, process, and analyze data streams at scale. It is commonly used for real-time analytics, log processing, and event-driven applications.

AWS Kinesis provides multiple services:

- **Kinesis Data Streams** – For ingesting and processing large streams of data.
- **Kinesis Data Firehose** – For delivering streaming data to AWS services like S3, Redshift, and Elasticsearch.
- **Kinesis Data Analytics** – For performing SQL queries on streaming data in real time.

---

1. Kinesis Configuration via AWS UI

1. Go to **AWS Management Console** → Open **Amazon Kinesis**
2. Click **Create data stream**
3. Enter **Stream name**
4. Choose **Shard count** (default: 1)
5. Click **Create data stream**

---

2. Kinesis Configuration via AWS CLI

Create a Kinesis Data Stream

aws kinesis create-stream --stream-name my-data-stream --shard-count 1

List Streams
aws kinesis list-streams

Delete Stream
aws kinesis delete-stream --stream-name my-data-stream

---

3. Kinesis Configuration via AWS CloudFormation

Create a kinesis-stream.yaml file:

yaml

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  MyKinesisStream:
    Type: AWS::Kinesis::Stream
    Properties:
      Name: my-data-stream
      ShardCount: 1
```

Deploy using CLI:

aws cloudformation create-stack --stack-name my-kinesis-stack --template-body file://kinesis-stream.yaml

---

4. Kinesis Configuration via Terraform

Create a main.tf file:

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_kinesis_stream" "example" {
  name        = "my-data-stream"
  shard_count = 1
}
```

Deploy using Terraform:

```
terraform init
terraform apply -auto-approve
```

---

5. Kinesis Configuration via Pulumi

Create a __main__.py file for Pulumi in Python:

```
python

import pulumi
import pulumi_aws as aws

stream = aws.kinesis.Stream("myDataStream", shard_count=1)

pulumi.export("stream_name", stream.name)
```

Deploy using Pulumi:

```
pulumi up
```

---

5. Data Pipeline: Data workflow orchestration.

Introduction to AWS Data Pipeline

AWS Data Pipeline is a managed service for **orchestrating** and **automating data movement** and **transformation** between AWS services (such as S3, RDS, and Redshift) and on-premise data sources. It helps schedule and manage data processing workflows without requiring manual intervention.

---

1. AWS Data Pipeline Configuration via AWS UI

1. Open **AWS Management Console** → Go to **AWS Data Pipeline**
2. Click **Create Pipeline**
3. Enter **Pipeline Name** and **Description**
4. Choose **Source** (S3, DynamoDB, RDS, etc.)
5. Choose **Destination** (S3, Redshift, etc.)
6. Set **Schedule** (e.g., daily, hourly)
7. Click **Activate**

---

2. AWS Data Pipeline Configuration via AWS CLI

Create a Data Pipeline

Create a JSON file (pipeline-definition.json) with the pipeline configuration:

json

```
{
  "objects": [
    {
```

```
    "id": "Default",
    "name": "MyDataPipeline",
    "schedule": {
      "type": "Schedule",
      "startDateTime": "2025-02-01T00:00:00",
      "period": "1 day"
    },
    "activity": {
      "type": "ShellCommandActivity",
      "scriptUri": "s3://my-bucket/script.sh",
      "runsOn": {
        "ref": "myInstance"
      }
    }
  },
  {
    "id": "myInstance",
    "type": "Ec2Resource",
    "instanceType": "t2.micro",
    "role": "DataPipelineDefaultRole"
  }
 ]
}
```

Create the pipeline using CLI:

aws datapipeline create-pipeline --name my-data-pipeline --unique-id my-data-pipeline

Upload the pipeline definition:

aws datapipeline put-pipeline-definition --pipeline-id <PIPELINE_ID> --pipeline-definition file://pipeline-definition.json

Activate the pipeline:

aws datapipeline activate-pipeline --pipeline-id <PIPELINE_ID>

---

3. AWS Data Pipeline Configuration via CloudFormation

Create a data-pipeline.yaml file:

yaml

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  MyDataPipeline:
    Type: "AWS::DataPipeline::Pipeline"
    Properties:
      Name: "MyDataPipeline"
      Description: "Pipeline for moving data from S3 to Redshift"
      Activate: true
      PipelineObjects:
        - Id: "Default"
          Name: "MyDataPipeline"
          Fields:
            - Key: "schedule"
              StringValue: "daily"
            - Key: "role"
              StringValue: "DataPipelineDefaultRole"
```

Deploy using CLI:

aws cloudformation create-stack --stack-name my-data-pipeline --template-body file://data-pipeline.yaml

---

4. AWS Data Pipeline Configuration via Terraform

Create a main.tf file:

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_datapipeline_pipeline" "example" {
  name        = "my-data-pipeline"
  description = "Pipeline for moving data"

  parameter_object {
    id = "Default"
    attributes {
      key   = "schedule"
      value = "daily"
    }
  }

  activate = true
}
```

Deploy using Terraform:

```
terraform init
terraform apply -auto-approve
```

---

5. AWS Data Pipeline Configuration via Pulumi

Create a __main__.py file for Pulumi in Python:

python

```
import pulumi
import pulumi_aws as aws

pipeline = aws.datapipeline.Pipeline("myDataPipeline",
    description="Pipeline for moving data from S3 to Redshift",
    activate=True
)

pulumi.export("pipeline_name", pipeline.name)
```

Deploy using Pulumi:

pulumi up

---

5. OpenSearch Service: Search and analytics.

Amazon OpenSearch Service: Search and Analytics

Amazon OpenSearch Service (formerly Amazon Elasticsearch Service) is a fully managed service that makes it easy to deploy, operate, and scale OpenSearch clusters. It is used for search, log analytics, observability, and real-time application monitoring. OpenSearch Service integrates with Kibana (OpenSearch Dashboards) for visualization and supports security, machine learning, and auto-scaling.

Configuration Methods for Amazon OpenSearch Service

1. AWS Management Console (UI)

- Navigate to **Amazon OpenSearch Service** in the AWS console.

- Click **Create domain** and choose **deployment type** (Production, Dev, or Testing).
- Select the **engine version** (OpenSearch or legacy Elasticsearch).
- Configure **instance type, storage, and network settings**.
- Enable security features such as fine-grained access control.
- Click **Create**, and AWS will provision the OpenSearch domain.

---

2. AWS CLI

```
aws opensearch create-domain \
  --domain-name my-opensearch-domain \
  --engine-version OpenSearch_2.9 \
  --cluster-config InstanceType=m6g.large.search,InstanceCount=2
```

3. AWS CloudFormation

yaml

```
Resources:
  OpenSearchDomain:
    Type: "AWS::OpenSearchService::Domain"
    Properties:
      DomainName: "my-opensearch-domain"
      EngineVersion: "OpenSearch_2.9"
      ClusterConfig:
        InstanceType: "m6g.large.search"
        InstanceCount: 2
      EBSOptions:
        EBSEnabled: true
        VolumeSize: 10
```

4. Terraform

hcl

```hcl
resource "aws_opensearch_domain" "example" {
  domain_name    = "my-opensearch-domain"
  engine_version = "OpenSearch_2.9"

  cluster_config {
    instance_type  = "m6g.large.search"
    instance_count = 2
  }

  ebs_options {
    ebs_enabled = true
    volume_size = 10
  }
}
```

5. Pulumi (Python)

python

```python
import pulumi
import pulumi_aws as aws

opensearch_domain = aws.opensearch.Domain("myOpensearchDomain",
    domain_name="my-opensearch-domain",
    engine_version="OpenSearch_2.9",
    cluster_config={
        "instance_type": "m6g.large.search",
        "instance_count": 2
    },
    ebs_options={
        "ebs_enabled": True,
        "volume_size": 10
    })
```

pulumi.export("opensearch_endpoint", opensearch_domain.endpoint)

Machine Learning

1. SageMaker: Build, train, and deploy ML models.

Amazon SageMaker: Introduction and Configuration Guide

Amazon SageMaker is a fully managed service that enables developers and data scientists to build, train, and deploy machine learning (ML) models quickly. It provides a broad set of features, including Jupyter notebooks, automatic model tuning, built-in algorithms, and integration with various AWS services like S3, Lambda, and IAM.

---

1. Configuring Amazon SageMaker via AWS UI

1. **Sign in to AWS Console** and navigate to **Amazon SageMaker**.
2. Click **Create Notebook Instance** (for development) or **Create Training Job** (for training).
3. Configure the instance type (e.g., ml.t2.medium).
4. Choose an IAM role with the necessary permissions.
5. Set up storage (Amazon S3 bucket for data).
6. Click **Create Notebook** or **Train Model**.
7. After training, go to **Inference → Create Endpoint** for deployment.

---

2. Configuring SageMaker using AWS CLI

Create a Notebook Instance

aws sagemaker create-notebook-instance \

```
    --notebook-instance-name my-notebook \
    --instance-type ml.t2.medium \
    --role-arn
arn:aws:iam::123456789012:role/service-role/AmazonSageMaker-ExecutionRole
```

Train a Model

```
aws sagemaker create-training-job \
    --training-job-name my-training-job \
    --algorithm-specification TrainingImage=IMAGE_URI,TrainingInputMode=File
\
    --role-arn
arn:aws:iam::123456789012:role/service-role/AmazonSageMaker-ExecutionRole \
    --resource-config
InstanceType=ml.m5.large,InstanceCount=1,VolumeSizeInGB=50 \
    --output-data-config S3OutputPath=s3://my-bucket/output/ \
    --stopping-condition MaxRuntimeInSeconds=3600
```

Deploy the Model

```
aws sagemaker create-endpoint \
    --endpoint-name my-endpoint \
    --endpoint-config-name my-endpoint-config
```

---

3. Configuring SageMaker using AWS CloudFormation

CloudFormation Template (YAML)

yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  MySageMakerNotebook:
    Type: "AWS::SageMaker::NotebookInstance"
```

```
  Properties:
    NotebookInstanceName: "my-notebook"
    InstanceType: "ml.t2.medium"
    RoleArn: "arn:aws:iam::123456789012:role/SageMakerRole"
    DirectInternetAccess: "Enabled"
```

Deploy the stack:

```
aws cloudformation create-stack --stack-name sagemaker-stack --template-body file://sagemaker-template.yaml
```

---

4. Configuring SageMaker using Terraform

Terraform Configuration

```
hcl

provider "aws" {
  region = "us-east-1"
}

resource "aws_sagemaker_notebook_instance" "notebook" {
  name          = "my-notebook"
  instance_type = "ml.t2.medium"
  role_arn      = "arn:aws:iam::123456789012:role/SageMakerRole"
}
```

Apply the configuration:

```
terraform init
terraform apply -auto-approve
```

---

5. Configuring SageMaker using Pulumi (Python)

Pulumi Python Code

python

```
import pulumi
import pulumi_aws as aws

sagemaker_role = aws.iam.Role("SageMakerRole",
    assume_role_policy="""{
        "Version": "2012-10-17",
        "Statement": [{
            "Effect": "Allow",
            "Principal": {"Service": "sagemaker.amazonaws.com"},
            "Action": "sts:AssumeRole"
        }]
    }""")

notebook = aws.sagemaker.NotebookInstance("myNotebook",
    instance_type="ml.t2.medium",
    role_arn=sagemaker_role.arn)

pulumi.export("notebook_name", notebook.name)
```

Deploy with Pulumi:

```
pulumi up
```

---

2. Rekognition: Image and video analysis.

Introduction to Amazon Rekognition

Amazon Rekognition is an AI service that provides image and video analysis capabilities, such as object detection, face recognition, text detection, and moderation. It uses deep learning to analyze images and videos, making it useful for security, content moderation, and automation applications.

---

1. Configuring Rekognition via AWS UI

1. **Sign in to AWS Console**
2. Navigate to **Amazon Rekognition**
3. Choose **Try Rekognition** to test images or videos
4. Enable Rekognition in IAM roles for applications that need access
5. Integrate Rekognition with S3, Lambda, or other AWS services

---

2. Configuring Rekognition via AWS CLI

Enable Rekognition for an AWS Account

aws rekognition create-project --project-name my-rekognition-project

Detect Faces in an Image

aws rekognition detect-faces --image "S3Object={Bucket=my-bucket,Name=image.jpg}" --attributes "ALL"

Detect Text in an Image:

aws rekognition detect-text --image "S3Object={Bucket=my-bucket,Name=text-image.jpg}"

---

3. Configuring Rekognition with AWS CloudFormation

Create a CloudFormation template (rekognition.yml):

yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  RekognitionRole:
    Type: AWS::IAM::Role
    Properties:
      RoleName: RekognitionServiceRole
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: rekognition.amazonaws.com
            Action: 'sts:AssumeRole'
      Policies:
        - PolicyName: RekognitionPolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action: rekognition:*
                Resource: "*"
```

Deploy using:

aws cloudformation create-stack --stack-name rekognition-stack --template-body file://rekognition.yml

---

4. Configuring Rekognition with Terraform

Create a Terraform file (rekognition.tf):

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_iam_role" "rekognition_role" {
  name = "rekognition_role"

  assume_role_policy = <<POLICY
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "rekognition.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
POLICY
}

resource "aws_iam_role_policy_attachment" "rekognition_policy" {
  role       = aws_iam_role.rekognition_role.name
  policy_arn = "arn:aws:iam::aws:policy/AmazonRekognitionFullAccess"
}
```

Deploy using:

```
terraform init
terraform apply
```

---

5. Configuring Rekognition with Pulumi (Python)

Install Pulumi AWS SDK:

pip install pulumi aws

Create a Pulumi file (__main__.py):

python

```
import pulumi
import pulumi_aws as aws

rekognition_role = aws.iam.Role("rekognitionRole",
    assume_role_policy="""{
      "Version": "2012-10-17",
      "Statement": [{
          "Effect": "Allow",
          "Principal": {"Service": "rekognition.amazonaws.com"},
          "Action": "sts:AssumeRole"
      }]
    }""")

rekognition_policy = aws.iam.RolePolicyAttachment("rekognitionPolicy",
    role=rekognition_role.name,
    policy_arn="arn:aws:iam::aws:policy/AmazonRekognitionFullAccess")

pulumi.export("rekognitionRoleName", rekognition_role.name)
```

Deploy using:

pulumi up

---

3. Comprehend: Natural language processing.

Introduction to Amazon Comprehend

Amazon Comprehend is a Natural Language Processing (NLP) service that uses machine learning to extract insights from text. It can perform sentiment analysis, entity recognition, key phrase extraction, topic modeling, and language detection. It helps automate text analysis for business applications, customer feedback, and document processing.

---

1. Configuring Amazon Comprehend via AWS UI

1. **Sign in to AWS Console**
2. Navigate to **Amazon Comprehend**
3. Click **Launch Comprehend**
4. Select an analysis type (Sentiment Analysis, Key Phrases, Entities, etc.)
5. Upload or enter text to analyze
6. Review results and integrate with applications

---

2. Configuring Amazon Comprehend via AWS CLI

Detect Sentiment in a Text String

aws comprehend detect-sentiment --language-code "en" --text "I love using AWS services!"

Detect Entities in a Text String

aws comprehend detect-entities --language-code "en" --text "Amazon is headquartered in Seattle."

Detect Key Phrases in a Text String

aws comprehend detect-key-phrases --language-code "en" --text "Machine learning is a part of artificial intelligence."

---

3. Configuring Amazon Comprehend with AWS CloudFormation

Create a CloudFormation template (comprehend.yml):

yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  ComprehendRole:
    Type: AWS::IAM::Role
    Properties:
      RoleName: ComprehendServiceRole
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: comprehend.amazonaws.com
            Action: 'sts:AssumeRole'
      Policies:
        - PolicyName: ComprehendPolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
```

```
      - Effect: Allow
        Action: comprehend:*
        Resource: "*"
```

Deploy using:

aws cloudformation create-stack --stack-name comprehend-stack --template-body file://comprehend.yml

---

4. Configuring Amazon Comprehend with Terraform

Create a Terraform file (comprehend.tf):

h

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_iam_role" "comprehend_role" {
  name = "comprehend_role"

  assume_role_policy = <<POLICY
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "comprehend.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
```

```
  ]
}
POLICY
}

resource "aws_iam_role_policy_attachment" "comprehend_policy" {
  role       = aws_iam_role.comprehend_role.name
  policy_arn = "arn:aws:iam::aws:policy/ComprehendFullAccess"
}
```

Deploy using:

```
terraform init
terraform apply
```

---

5. Configuring Amazon Comprehend with Pulumi (Python)

Install Pulumi AWS SDK:

sh

```
pip install pulumi aws
```

Create a Pulumi file (__main__.py):

python

```
import pulumi
import pulumi_aws as aws

comprehend_role = aws.iam.Role("comprehendRole",
    assume_role_policy="""{
```

```
    "Version": "2012-10-17",
    "Statement": [{
      "Effect": "Allow",
      "Principal": {"Service": "comprehend.amazonaws.com"},
      "Action": "sts:AssumeRole"
    }]
  }""")

comprehend_policy = aws.iam.RolePolicyAttachment("comprehendPolicy",
  role=comprehend_role.name,
  policy_arn="arn:aws:iam::aws:policy/ComprehendFullAccess")

pulumi.export("comprehendRoleName", comprehend_role.name)
```

Deploy using:

```
pulumi up
```

---

4. Polly: Text-to-speech.

Introduction to Amazon Polly

Amazon Polly is a service that turns text into lifelike speech using deep learning models. It supports multiple languages and voices, making it ideal for building applications that require speech synthesis. Polly can be used for creating audio files, adding voice to chatbots, and integrating speech capabilities into IoT devices, mobile apps, and more.

---

1. Configuring Amazon Polly via AWS UI

1. **Sign in to AWS Console**

2. Navigate to **Amazon Polly**
3. Choose **Text-to-Speech**
4. Enter the text you want to convert to speech
5. Select the desired language, voice, and speech format (MP3, OGG, etc.)
6. Click **Synthesize Speech**
7. Download the generated speech audio file

---

2. Configuring Amazon Polly via AWS CLI

Synthesize Speech from Text

aws polly synthesize-speech --text "Hello, welcome to AWS Polly!" --output-format mp3 --voice-id Joanna --language-code en-US --file-name output.mp3

List Available Voices

aws polly describe-voices --language-code en-US

Save Speech to an Audio File

aws polly synthesize-speech --text "Welcome to Amazon Polly" --voice-id "Joanna" --output-format "mp3" --output "welcome.mp3"

---

3. Configuring Amazon Polly with AWS CloudFormation

Create a CloudFormation template (polly.yml):

yaml

AWSTemplateFormatVersion: '2010-09-09'
Resources:

```
  PollyRole:
    Type: AWS::IAM::Role
    Properties:
      RoleName: PollyServiceRole
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: polly.amazonaws.com
            Action: 'sts:AssumeRole'
      Policies:
        - PolicyName: PollyPolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action: polly:*
                Resource: "*"
```

Deploy using:

aws cloudformation create-stack --stack-name polly-stack --template-body file://polly.yml

---

4. Configuring Amazon Polly with Terraform

Create a Terraform file (polly.tf):

hcl

```
provider "aws" {
  region = "us-east-1"
```

```
}

resource "aws_iam_role" "polly_role" {
  name = "polly_role"

  assume_role_policy = <<POLICY
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "polly.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
POLICY
}

resource "aws_iam_role_policy_attachment" "polly_policy" {
  role       = aws_iam_role.polly_role.name
  policy_arn = "arn:aws:iam::aws:policy/PollyFullAccess"
}
```

Deploy using:

```
terraform init
terraform apply
```

---

5. Configuring Amazon Polly with Pulumi (Python)

Install Pulumi AWS SDK:

pip install pulumi aws

Create a Pulumi file (__main__.py):

python

```
import pulumi
import pulumi_aws as aws

polly_role = aws.iam.Role("pollyRole",
   assume_role_policy="""{
     "Version": "2012-10-17",
     "Statement": [{
        "Effect": "Allow",
        "Principal": {"Service": "polly.amazonaws.com"},
        "Action": "sts:AssumeRole"
     }]
   }""")

polly_policy = aws.iam.RolePolicyAttachment("pollyPolicy",
   role=polly_role.name,
   policy_arn="arn:aws:iam::aws:policy/PollyFullAccess")

pulumi.export("pollyRoleName", polly_role.name)
```

Deploy using

pulumi up

---

5. Translate: Language translation.

AWS Introduction & Configuration with Various Tools

AWS (Amazon Web Services) offers a comprehensive suite of cloud services, providing flexibility for managing infrastructure. You can configure and manage your AWS environment using different tools like the AWS UI, CLI, CloudFormation, Terraform, and Pulumi.

---

1. **AWS UI Configuration**:
    - **Step 1**: Log in to the AWS Management Console.
    - **Step 2**: Navigate to the desired service (e.g., EC2, S3, Lambda).
    - **Step 3**: Follow the on-screen steps to configure the service.

---

2. **AWS CLI Configuration**:

Step 1: Install AWS CLI:
pip install awscli

Step 2: Configure AWS credentials:
aws configure

Step 3: Use CLI commands to manage AWS services. Example to list S3 buckets:
aws s3 ls

---

3. **AWS CloudFormation Configuration**:
    - **Step 1**: Create a template.yml file to define your AWS resources.

Step 2: Deploy the CloudFormation stack:

aws cloudformation create-stack --stack-name MyStack --template-body file://template.yml

- **Terraform Configuration**:

Step 1: Install Terraform:
brew install terraform

  - **Step 2**: Create a main.tf file defining the AWS infrastructure.

Step 3: Initialize and apply the Terraform plan:
terraform init
terraform apply

- **Pulumi Configuration**:

Step 1: Install Pulumi:
brew install
pulumi

  - **Step 2**: Create a Pulumi project with a index.ts (for TypeScript) or index.py (for Python).

Step 3: Deploy with Pulumi:
pulumi up

---

6. Lex: Build conversational interfaces.

AWS Lex: Build Conversational Interfaces

AWS Lex is a service provided by Amazon Web Services that enables you to build conversational interfaces into applications using voice and text. It helps in developing chatbots and voice-enabled applications seamlessly, leveraging automatic speech recognition (ASR) and natural language processing (NLP).

Configuration Methods for AWS Lex:

---

1. **AWS UI (Console):**

To configure AWS Lex through the UI:

1. **Sign in to the AWS Management Console**.
2. **Navigate to Amazon Lex** under the "AI & Machine Learning" section.
3. **Create a new bot** by choosing a pre-built template or starting from scratch.
4. Define intents, sample phrases, and slots (optional).
5. Set up **Lambda functions** to handle fulfillment.
6. Configure **permissions**, and test the bot within the console.

---

2. AWS CLI:

With AWS CLI, you can create and manage Lex bots using commands.

Create Bot:

```
aws lex-models create-bot \
  --name "BookHotel" \
  --locale "en-US" \
  --intents file://intents.json \
  --childDirected false
```

Create Intent:
aws lex-models create-intent \
  --name "BookHotelIntent" \
  --sample-utterances "I want to book a hotel" "Book a hotel room"

> You can also use update-bot and delete-bot commands for modifying and removing bots.

---

3. CloudFormation:

CloudFormation allows you to define the infrastructure for Lex in a declarative JSON or YAML format.

Example YAML configuration for AWS Lex bot:

yaml

```
Resources:
  LexBot:
    Type: AWS::Lex::Bot
    Properties:
      Name: "BookHotelBot"
      Description: "Bot to book hotels"
      Locale: "en-US"
      Intents:
        - IntentName: "BookHotelIntent"
          IntentVersion: "1"
      VoiceSettings:
        VoiceId: "Joanna"
```

Deploy using the following command:

aws cloudformation deploy --template-file lex-bot.yaml --stack-name LexBotStack

---

4. Terraform:

Terraform lets you manage AWS Lex resources using Infrastructure as Code (IaC).

Example Terraform configuration:

hcl

```
resource "aws_lex_bot" "book_hotel_bot" {
  name        = "BookHotelBot"
  description = "Bot to book hotels"
  locale      = "en-US"

  intent {
    name          = "BookHotelIntent"
    intent_version = "1"
  }
}
```

Apply configuration:

```
terraform init
terraform apply
```

---

5. Pulumi:

Pulumi also supports AWS Lex, and you can define and deploy Lex bots using code.

Example Pulumi (TypeScript) configuration:

typescript

```
import * as aws from "@pulumi/aws";

const bookHotelBot = new aws.lex.Bot("bookHotelBot", {
  name: "BookHotelBot",
  description: "Bot for booking hotel",
  locale: "en-US",
  intents: [{
    intentName: "BookHotelIntent",
    intentVersion: "1",
  }],
});

export const botName = bookHotelBot.name;
```

Deploy with:

```
pulumi up
```

---

7. Personalize: Personalization and recommendations.

AWS Personalize is a machine learning service by AWS that helps developers build personalized recommendations for their users without the need for ML expertise. It uses advanced algorithms to deliver personalized product, content, or search recommendations based on user preferences, past behaviors, and more.

---

1. Configuration Using AWS UI

1. **Create a Personalize Dataset Group**:
    - Log into the AWS Management Console.
    - Navigate to **Amazon Personalize** under the AI & Machine Learning section.

- Click on **Create dataset group**.
- Choose the **use case** (e.g., Personalized Ranking or User Personalization).
- Provide a **Dataset group name** and configure it.

2. **Create and Import Datasets**:
   - After the dataset group is created, click on **Create dataset**.
   - Choose the dataset type (e.g., Interactions, Users, or Items).
   - Import data via an S3 bucket.
3. **Create a Solution and Campaign**:
   - After uploading the dataset, create a **Solution** using one of the predefined algorithms.
   - Build the **Campaign** for real-time recommendation delivery.

---

2. Configuration Using AWS CLI

Create Dataset Group:
aws personalize create-dataset-group --name my-dataset-group

Create Datasets:
aws personalize create-dataset --dataset-group-arn <dataset-group-arn> --name my-dataset --schema-arn <schema-arn> --dataset-type Interactions

Import Data:
aws personalize create-dataset-import-job --dataset-arn <dataset-arn> --job-name my-import-job --data-source dataLocation=<s3-uri> --role-arn <iam-role-arn>

Create Solution:
aws personalize create-solution --name my-solution --dataset-group-arn <dataset-group-arn> --recipe ARN --perform-hpo

Create Campaign:
aws personalize create-campaign --name my-campaign --solution-arn <solution-arn> --min-provisioned-transaction-per-second 1

---

3. Configuration Using CloudFormation

yaml

```
Resources:
  MyDatasetGroup:
    Type: AWS::Personalize::DatasetGroup
    Properties:
      Name: my-dataset-group

  MyDataset:
    Type: AWS::Personalize::Dataset
    Properties:
      DatasetGroupArn: !GetAtt MyDatasetGroup.Arn
      Name: my-dataset
      DatasetType: Interactions
      SchemaArn: arn:aws:personalize:region:account-id:schema/my-schema

  MySolution:
    Type: AWS::Personalize::Solution
    Properties:
      DatasetGroupArn: !Ref MyDatasetGroup
      Name: my-solution
      RecipeArn: arn:aws:personalize:region:account-id:recipe/recipe-arn
```

---

4. Configuration Using Terraform

hcl

```
resource "aws_personalize_dataset_group" "example" {
```

```
  name = "my-dataset-group"
}

resource "aws_personalize_dataset" "example" {
  dataset_group_arn = aws_personalize_dataset_group.example.arn
  name              = "my-dataset"
  dataset_type      = "Interactions"
  schema_arn        =
"arn:aws:personalize:region:account-id:schema/my-schema"
}

resource "aws_personalize_solution" "example" {
  name              = "my-solution"
  dataset_group_arn  = aws_personalize_dataset_group.example.arn
  recipe_arn        =
"arn:aws:personalize:region:account-id:recipe/recipe-arn"
  perform_hpo       = true
}
```

---

5. Configuration Using Pulumi (with TypeScript)

typescript

```
import * as aws from "@pulumi/aws";

// Create Dataset Group
const datasetGroup = new
aws.personalize.DatasetGroup("my-dataset-group", {
    name: "my-dataset-group"
});

// Create Dataset
const dataset = new aws.personalize.Dataset("my-dataset", {
```

```
    datasetGroupArn: datasetGroup.arn,
    name: "my-dataset",
    datasetType: "Interactions",
    schemaArn: "arn:aws:personalize:region:account-id:schema/my-schema",
});

// Create Solution
const solution = new aws.personalize.Solution("my-solution", {
    datasetGroupArn: datasetGroup.arn,
    name: "my-solution",
    recipeArn: "arn:aws:personalize:region:account-id:recipe/recipe-arn",
    performHpo: true,
});
```

---

8. Forecast: Time-series forecasting.

Introduction: Time-series forecasting involves predicting future values based on previously observed data, commonly applied in areas such as finance, weather prediction, and inventory management. AWS offers a variety of tools to configure and manage resources for time-series forecasting, including through the AWS Management Console (UI), AWS CLI, CloudFormation, Terraform, and Pulumi.

---

1. AWS UI Configuration: To get started with time-series forecasting in AWS using the UI, you can use Amazon Forecast, a fully managed service for time-series forecasting. Here's a basic configuration process:

- **Create a dataset group**: In the Amazon Forecast Console, go to the "Dataset Groups" section and create a new dataset group.
- **Add data**: Import time-series data such as historical data or related data through CSV or S3.

- **Create a predictor**: Once your dataset is in place, you can create a predictor with a forecast model, specifying attributes such as frequency (daily, monthly, etc.), the time-series length, and more.

---

2. AWS CLI Configuration: To configure time-series forecasting using the AWS CLI, follow these steps:

Create a Dataset Group:
aws forecast create-dataset-group --dataset-group-name "my-dataset-group"

Import Data: Use the CLI to import historical time-series data:
aws forecast create-dataset-import-job --dataset-import-job-name "my-import-job" --dataset-arn "dataset_arn" --data-source "S3_URI"

Create a Predictor:
aws forecast create-predictor --predictor-name "my-predictor" --dataset-group-arn "dataset_group_arn"

---

3. AWS CloudFormation Configuration: To use AWS CloudFormation for time-series forecasting, define the resources like datasets and predictors in a YAML/JSON template:

yaml

```
Resources:
  MyDatasetGroup:
    Type: AWS::Forecast::DatasetGroup
    Properties:
      DatasetGroupName: "MyForecastGroup"

  MyPredictor:
    Type: AWS::Forecast::Predictor
    Properties:
```

```
    PredictorName: "MyPredictor"
    DatasetGroupArn: !Ref MyDatasetGroup
```

You can then deploy this CloudFormation stack with:

```
aws cloudformation create-stack --stack-name "MyForecastStack" --template-body file://forecast-template.yaml
```

---

4. Terraform Configuration: Using Terraform, you can configure AWS Forecast resources. Here's a simple example:

```
hcl

resource "aws_forecast_dataset_group" "my_dataset_group" {
  name = "my-forecast-group"
}

resource "aws_forecast_predictor" "my_predictor" {
  name              = "my-predictor"
  dataset_group_arn  = aws_forecast_dataset_group.my_dataset_group.arn
  algorithm_arn      = "arn:aws:forecast:::algorithm/ARIMA"
}
```

To apply the configuration:

```
terraform apply
```

---

5. Pulumi Configuration: Pulumi allows defining resources using programming languages like TypeScript, Python, Go, or C#. Here’s a sample in TypeScript:

typescript

```
import * as aws from "@pulumi/aws";

// Create a Dataset Group
const datasetGroup = new aws.forecast.DatasetGroup("myDatasetGroup", {
    datasetGroupName: "my-forecast-group",
});

// Create a Predictor
const predictor = new aws.forecast.Predictor("myPredictor", {
    predictorName: "my-predictor",
    datasetGroupArn: datasetGroup.arn,
    algorithmArn: "arn:aws:forecast:::algorithm/ARIMA",
});
```

Run the following to deploy the stack:

```
pulumi up
```

Management & Governance

1. CloudWatch: Monitoring and logging.

AWS CloudWatch: Monitoring and Logging

Amazon CloudWatch is a monitoring and observability service for AWS cloud resources and applications. It provides metrics, logs, and events, enabling users to monitor and gain insights into their AWS resources, applications, and services. CloudWatch helps with the real-time monitoring of system performance, application logs, and custom metrics. It integrates with a variety of AWS services to offer automated monitoring and alerting.

---

1. AWS UI (Management Console)

To configure **CloudWatch** using the AWS UI:

- Go to the **AWS Management Console**.
- Navigate to **CloudWatch** under the "Services" menu.
- Here, you can create **Alarms**, set up **Log Groups** and **Log Streams**, and manage **Metrics**.
- You can also create **CloudWatch Dashboards** to visualize your resources and monitor their health.

---

2. AWS CLI

To configure **CloudWatch** using the AWS CLI:

- Ensure AWS CLI is installed and configured.

Create an alarm for an EC2 instance's CPU utilization:
aws cloudwatch put-metric-alarm --alarm-name "HighCPUAlarm" --metric-name CPUUtilization --namespace AWS/EC2 --statistic Average --period 300 --threshold 80 --comparison-operator GreaterThanOrEqualToThreshold --dimension Name=InstanceId,Value=i-12345678 --evaluation-periods 2 --alarm-actions arn:aws:sns:us-west-2:123456789012:MyTopic

Log an event using the CLI:
aws logs create-log-group --log-group-name my-log-group
aws logs create-log-stream --log-group-name my-log-group --log-stream-name my-log-stream

---

3. AWS CloudFormation

To configure **CloudWatch** using **CloudFormation**:

- CloudFormation templates can automate the setup of CloudWatch resources like alarms, log groups, and dashboards.

Example for creating a **CloudWatch Alarm**:

yaml

```
Resources:
  HighCPUAlarm:
    Type: AWS::CloudWatch::Alarm
    Properties:
      AlarmName: "HighCPUAlarm"
      MetricName: CPUUtilization
      Namespace: AWS/EC2
      Statistic: Average
      Period: 300
      Threshold: 80
      ComparisonOperator: GreaterThanOrEqualToThreshold
      EvaluationPeriods: 2
      Dimensions:
        - Name: InstanceId
          Value: i-12345678
      AlarmActions:
        - arn:aws:sns:us-west-2:123456789012:MyTopic
```

---

4. Terraform

To configure **CloudWatch** using **Terraform**:

- Use the **aws_cloudwatch_metric_alarm** resource to define alarms, and **aws_cloudwatch_log_group** for log groups.

Example Terraform configuration:

hcl

```
resource "aws_cloudwatch_log_group" "my_log_group" {
  name = "/aws/lambda/my-log-group"
}

resource "aws_cloudwatch_metric_alarm" "high_cpu_alarm" {
  alarm_name          = "HighCPUAlarm"
  comparison_operator = "GreaterThanOrEqualToThreshold"
  evaluation_periods  = 2
  metric_name         = "CPUUtilization"
  namespace           = "AWS/EC2"
  period              = 300
  statistic           = "Average"
  threshold           = 80
  dimensions = {
    InstanceId = "i-12345678"
  }
  alarm_actions = ["arn:aws:sns:us-west-2:123456789012:MyTopic"]
}
```

---

5. Pulumi

To configure **CloudWatch** using **Pulumi**:

- Pulumi allows you to define CloudWatch resources in modern programming languages like TypeScript, Python, and Go.

Example in **TypeScript**:

typescript

```
import * as aws from "@pulumi/aws";

// Create a CloudWatch log group
const logGroup = new aws.cloudwatch.LogGroup("my-log-group");

// Create a CloudWatch alarm
const cpuAlarm = new aws.cloudwatch.MetricAlarm("highCPUAlarm", {
    comparisonOperator: "GreaterThanOrEqualToThreshold",
    evaluationPeriods: 2,
    metricName: "CPUUtilization",
    namespace: "AWS/EC2",
    period: 300,
    statistic: "Average",
    threshold: 80,
    dimensions: {
        InstanceId: "i-12345678",
    },
    alarmActions: ["arn:aws:sns:us-west-2:123456789012:MyTopic"],
});
```

---

2. CloudFormation: Infrastructure as code.

Introduction to Infrastructure as Code (IaC):

Infrastructure as Code (IaC) is the process of managing and provisioning computing infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools. Using IaC, you can automate your infrastructure setup, reduce human errors, and ensure consistency across environments.

---

1. AWS UI (Management Console):

To configure AWS CloudFormation using the AWS UI:

1. **Open CloudFormation**:
    - Go to the AWS Management Console.
    - In the search bar, type "CloudFormation" and select it.
2. **Create a Stack**:
    - Click on **Create Stack**.
    - Choose **Template Source** (you can use an existing template or upload your own).
    - If uploading a file, choose **Upload a template file**, then upload your YAML or JSON template.
3. **Specify Stack Details**:
    - Provide the **stack name** and fill out any parameters needed.
4. **Configure Stack Options**:
    - Set up options such as tags, IAM roles, permissions, etc.
5. **Review and Create**:
    - Review your settings and click **Create Stack**.

---

2. AWS CLI:

To configure AWS CloudFormation using AWS CLI:

1. **Install AWS CLI** if you haven't already.

Set up your AWS credentials:
nginx
aws configure

Create a Stack: Use the following command to create a stack from a CloudFormation template:

aws cloudformation create-stack --stack-name <stack-name> --template-body file://template.yaml

Update a Stack (if required):
aws cloudformation update-stack --stack-name <stack-name> --template-body file://template.yaml

Delete a Stack:
aws cloudformation delete-stack --stack-name <stack-name>

---

3. Terraform:

To configure AWS CloudFormation using Terraform:

1. **Install Terraform** if you haven't already.

Set up AWS provider in your main.tf file:
hcl

```
provider "aws" {
  region = "us-east-1"
}
```

Create CloudFormation Stack Resource:

```
resource "aws_cloudformation_stack" "example" {
  name = "example-stack"
  template_body = file("template.yaml")

  parameters = {
    Param1 = "value1"
  }
```

}

2. **Run Terraform Commands**:

Initialize Terraform:
csharp

terraform init

Apply the configuration:
nginx
terraform apply

Destroy the stack (if needed):
nginx
terraform destroy

---

4. Pulumi:

To configure AWS CloudFormation using Pulumi:

1. **Install Pulumi** if you haven't already.

Set up AWS provider in your index.ts (TypeScript) or index.py (Python) file:
Example in TypeScript:
typescript

```
import * as aws from "@pulumi/aws";

const stack = new aws.cloudformation.Stack("example", {
   templateBody: fs.readFileSync("template.yaml", "utf-8"),
});
```

2. **Run Pulumi Commands**:

Initialize Pulumi:
pulumi new aws-typescript

Preview the changes:
nginx
pulumi preview

Apply the changes:
nginx
pulumi up

Destroy the stack (if needed):
nginx
pulumi destroy

---

3. CloudTrail: Governance and compliance.

AWS CloudTrail: Governance and Compliance

Introduction:
AWS CloudTrail is a service that enables governance, compliance, and operational and risk auditing of your AWS account. It records API calls and user activity, providing visibility into actions taken within AWS. CloudTrail helps organizations meet security and compliance requirements by logging and storing event history for analysis and threat detection.

---

1. Configure CloudTrail via AWS UI:

1. Sign in to the AWS Management Console.
2. Navigate to **CloudTrail** from the AWS Services menu.
3. Click **Create Trail** and provide a trail name.
4. Choose **'Create a new S3 bucket'** or select an existing bucket for log storage.
5. (Optional) Enable **CloudWatch Logs** for real-time monitoring.
6. Choose **Event Types** (Management, Data, or Insights events).
7. Click **Create Trail** to complete the setup.

---

2. Configure CloudTrail using AWS CLI:

Enable a CloudTrail trail that logs all API activities to an S3 bucket.

```
aws cloudtrail create-trail \
  --name MyTrail \
  --s3-bucket-name my-cloudtrail-logs \
  --is-multi-region-trail
```

Enable logging:

```
aws cloudtrail start-logging --name MyTrail
```

Verify the status:

```
aws cloudtrail describe-trails
```

---

3. Configure CloudTrail using AWS CloudFormation:

Create a cloudtrail.yml file with the following template:

yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Resources:
  MyTrail:
    Type: AWS::CloudTrail::Trail
    Properties:
      TrailName: MyCloudTrail
      S3BucketName: my-cloudtrail-logs
      IsMultiRegionTrail: true
      EnableLogFileValidation: true
      IncludeGlobalServiceEvents: true
```

Deploy using AWS CLI:

```
aws cloudformation create-stack --stack-name CloudTrailStack --template-body file://cloudtrail.yml
```

---

4. Configure CloudTrail using Terraform:

Create a cloudtrail.tf file:

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_s3_bucket" "trail_bucket" {
  bucket = "my-cloudtrail-logs"
}

resource "aws_cloudtrail" "trail" {
  name                          = "MyCloudTrail"
  s3_bucket_name                = aws_s3_bucket.trail_bucket.id
  include_global_service_events = true
  is_multi_region_trail         = true
```

}

Apply the configuration:

terraform init
terraform apply -auto-approve

---

5. Configure CloudTrail using Pulumi (Python):

Install Pulumi AWS package:

pip install pulumi pulumi-aws

Create a __main__.py file:

python

```
import pulumi
import pulumi_aws as aws

bucket = aws.s3.Bucket("cloudtrail-logs")

trail = aws.cloudtrail.Trail("myCloudTrail",
    s3_bucket_name=bucket.id,
    include_global_service_events=True,
    is_multi_region_trail=True)

pulumi.export("bucket_name", bucket.id)
pulumi.export("trail_name", trail.id)
```

Deploy using Pulumi CLI:

pulumi up

---

4. Config: Resource compliance tracking.

Introduction to AWS Resource Compliance Tracking

AWS resource compliance tracking ensures that cloud resources adhere to organizational policies, security standards, and best practices. AWS Config, AWS Security Hub, and AWS Audit Manager are commonly used to assess resource configurations and compliance with predefined rules.

---

1. Configuring AWS Resource Compliance Tracking

1. Using AWS Management Console (UI)

1. Navigate to **AWS Config** in the AWS Management Console.
2. Click **Get Started** and choose a **Recorder** to track configuration changes.
3. Select an **S3 bucket** for storing configuration logs.
4. Define **rules** based on AWS-managed or custom policies.
5. Enable **Amazon SNS notifications** for compliance alerts.
6. Review and click **Save Configuration**.

---

2. Using AWS CLI

Enable AWS Config and set up resource compliance tracking with CLI:

```
aws configservice put-configuration-recorder \
  --configuration-recorder name=default \
  --role-ARN arn:aws:iam::ACCOUNT_ID:role/AWSConfigRole

aws configservice put-delivery-channel \
```

```
  --delivery-channel file://delivery-channel.json

aws configservice start-configuration-recorder \
  --configuration-recorder-name default
```

Sample delivery-channel.json:

json

```
{
  "name": "default",
  "s3BucketName": "my-config-bucket",
  "snsTopicARN": "arn:aws:sns:us-east-1:123456789012:MySNSTopic"
}
```

---

3. Using AWS CloudFormation

A sample CloudFormation template to enable AWS Config and track compliance:

yaml

```
Resources:
  ConfigRecorder:
    Type: AWS::Config::ConfigurationRecorder
    Properties:
      Name: default
      RoleARN: arn:aws:iam::ACCOUNT_ID:role/AWSConfigRole

  DeliveryChannel:
    Type: AWS::Config::DeliveryChannel
    Properties:
      S3BucketName: my-config-bucket
```

```
    SnsTopicARN: arn:aws:sns:us-east-1:123456789012:MySNSTopic

  ConfigRule:
    Type: AWS::Config::ConfigRule
    Properties:
      ConfigRuleName: s3-bucket-public-read-prohibited
      Source:
        Owner: AWS
        SourceIdentifier: S3_BUCKET_PUBLIC_READ_PROHIBITED
```

---

4. Using Terraform

A Terraform configuration to enable AWS Config and compliance rules:

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_config_configuration_recorder" "config" {
  name     = "default"
  role_arn = "arn:aws:iam::ACCOUNT_ID:role/AWSConfigRole"
}

resource "aws_config_delivery_channel" "config" {
  s3_bucket_name = "my-config-bucket"
}

resource "aws_config_configuration_recorder_status" "config" {
  name       = aws_config_configuration_recorder.config.name
  is_enabled = true
}
```

---

5. Using Pulumi

Pulumi code (Python) to configure AWS Config tracking:

python

```
import pulumi
import pulumi_aws as aws

role = aws.iam.Role("awsConfigRole", assume_role_policy="""{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "sts:AssumeRole",
      "Principal": { "Service": "config.amazonaws.com" },
      "Effect": "Allow",
      "Sid": ""
    }
  ]
}""")

recorder = aws.cfg.ConfigurationRecorder("configRecorder",
    role_arn=role.arn)

delivery_channel = aws.cfg.DeliveryChannel("deliveryChannel",
    s3_bucket_name="my-config-bucket")

pulumi.export("recorder", recorder.name)
pulumi.export("delivery_channel", delivery_channel.name)
```

---

5. Trusted Advisor: Resource optimization and recommendations.

AWS **Trusted Advisor** is a service that provides real-time guidance to help you optimize your AWS environment by following best practices in cost optimization, security, fault tolerance, performance, and service limits. It analyzes your AWS resources and provides recommendations to improve efficiency, enhance security, and reduce costs.

Configuration Methods

1. AWS UI (Management Console)

- Sign in to the **AWS Management Console**.
- Navigate to **Trusted Advisor** from the **AWS Support** section.
- Review recommendations under different categories such as **Cost Optimization**, **Security**, **Performance**, **Reliability**, and **Service Limits**.
- Apply suggested optimizations based on provided insights.

---

2. AWS CLI (Command Line Interface)

Trusted Advisor can be accessed using the AWS CLI to retrieve recommendations.

Install and configure AWS CLI:
aws configure

List Trusted Advisor checks:
aws support describe-trusted-advisor-checks --language en

Retrieve Trusted Advisor check results:
aws support describe-trusted-advisor-check-result --check-id <CHECK_ID>

---

3. AWS CloudFormation

AWS CloudFormation does not directly configure Trusted Advisor but can be used to deploy infrastructure following best practices.

Example: Defining an **IAM role** with least privileges for cost optimization recommendations:

yaml

```
Resources:
  CostOptimizationRole:
    Type: AWS::IAM::Role
    Properties:
      RoleName: TrustedAdvisorCostRole
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Effect: Allow
            Principal:
              Service: "trustedadvisor.amazonaws.com"
            Action: "sts:AssumeRole"
```

---

4. Terraform Configuration

While Terraform does not configure Trusted Advisor directly, it can be used to set up AWS resources following Trusted Advisor recommendations.

Example: Enabling AWS Support API to fetch Trusted Advisor recommendations:

hcl

```
provider "aws" {
  region = "us-east-1"
}
```

```
resource "aws_iam_role" "trusted_advisor_role" {
  name = "TrustedAdvisorRole"
  assume_role_policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "trustedadvisor.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF
}
```

---

5. Pulumi Configuration

Pulumi can manage AWS resources by applying infrastructure best practices recommended by Trusted Advisor.

Example: Creating an IAM role for Trusted Advisor recommendations using Pulumi (Python):

python

```
import pulumi
import pulumi_aws as aws

trusted_advisor_role = aws.iam.Role("TrustedAdvisorRole",
    assume_role_policy="""{
```

```
        "Version": "2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Principal": { "Service": "trustedadvisor.amazonaws.com" },
                "Action": "sts:AssumeRole"
            }
        ]
    }"""
)

pulumi.export("role_name", trusted_advisor_role.name)
```

---

6. Control Tower: Multi-account governance.

AWS Control Tower: Multi-Account Governance

AWS Control Tower provides a managed service to set up and govern a secure, multi-account AWS environment following AWS best practices. It automates the setup of AWS Organizations, IAM, logging, and security policies, enabling centralized governance while allowing flexibility for different teams.

---

1. AWS Control Tower Configuration Using UI

1. Navigate to the **AWS Control Tower Console**.
2. Click **Set up Landing Zone** and choose a management account.
3. Configure organizational units (OUs), guardrails, and logging.
4. Enable AWS Config and AWS CloudTrail for auditing.
5. Review settings and launch the Control Tower setup.

---

2. AWS Control Tower Configuration Using AWS CLI

Ensure you have the AWS CLI installed and configured with appropriate permissions.

 # Enable AWS Control Tower
aws controltower enable-control-tower

 # List available controls
aws controltower list-controls --control-type "PREVENTIVE"

 # Register an organizational unit (OU)
aws controltower register-organizational-unit --organizational-unit-id <OU-ID>

---

3. AWS Control Tower Configuration Using CloudFormation

You can deploy a CloudFormation template to create AWS Organizations and apply policies.

yaml

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  Organization:
    Type: AWS::Organizations::Organization
    Properties:
      FeatureSet: "ALL"
  OU:
    Type: AWS::Organizations::OrganizationalUnit
    Properties:
      Name: "DevOU"
      ParentId: !GetAtt Organization.RootId
```

Deploy the stack using:

aws cloudformation create-stack --stack-name ControlTowerStack --template-body file://controltower.yaml

---

4. AWS Control Tower Configuration Using Terraform

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_organizations_organization" "main" {
  feature_set = "ALL"
}

resource "aws_organizations_organizational_unit" "dev" {
  name      = "DevOU"
  parent_id = aws_organizations_organization.main.roots[0].id
}
```

Apply the configuration:

terraform init
terraform apply -auto-approve

---

5. AWS Control Tower Configuration Using Pulumi

python

```
import pulumi
import pulumi_aws as aws
```

```
org = aws.organizations.Organization(feature_set="ALL")

dev_ou = aws.organizations.OrganizationalUnit("DevOU",
    name="DevOU",
    parent_id=org.roots[0].id)
```

Run Pulumi commands:

```
pulumi up
```

---

7. Organizations: Manage multiple AWS accounts.

AWS Organizations helps businesses manage and govern multiple AWS accounts from a central location. It provides consolidated billing, security policies, and service control policies (SCPs) to enforce governance across accounts. Organizations can be structured hierarchically using Organizational Units (OUs).

---

1. Configuring AWS Organizations via AWS UI

1. **Sign in** to the AWS Management Console with a management account.
2. Go to **AWS Organizations** and click **Create an Organization** (if not already created).
3. Enable **All Features** for full management capabilities.
4. Click **Add an AWS Account** to either:
    - **Create a new account**
    - **Invite an existing AWS account** using an email or AWS Account ID
5. Use **Service Control Policies (SCPs)** to restrict actions in member accounts.

---

2. Configuring AWS Organizations via AWS CLI

Ensure AWS CLI is installed and configured with necessary IAM permissions.

Create an Organization
aws organizations create-organization --feature-set ALL

Create an Organizational Unit (OU)
aws organizations create-organizational-unit --parent-id <ROOT_ID> --name "Development"

Create a New AWS Account
aws organizations create-account --email "newaccount@example.com" --account-name "DevAccount"

Attach SCP to OU or Account
aws organizations attach-policy --policy-id <POLICY_ID> --target-id <ACCOUNT_ID or OU_ID>

---

3. Configuring AWS Organizations with CloudFormation

CloudFormation templates can define AWS Organizations structure as Infrastructure as Code (IaC).

Example CloudFormation Template
yaml

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  MyOrganization:
    Type: "AWS::Organizations::Organization"
    Properties:
      FeatureSet: "ALL"
```

Deploy with AWS CLI:

aws cloudformation create-stack --stack-name my-org --template-body file://org.yml

---

4. Configuring AWS Organizations with Terraform

Terraform allows defining AWS Organizations in a declarative way.

Example Terraform Configuration

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_organizations_organization" "org" {
  feature_set = "ALL"
}

resource "aws_organizations_account" "dev_account" {
  name  = "DevAccount"
  email = "newaccount@example.com"
}
```

Deploy with Terraform:

terraform init
terraform apply -auto-approve

---

5. Configuring AWS Organizations with Pulumi

Pulumi uses programming languages like Python, TypeScript, or Go for AWS Organizations setup.

Example Pulumi (TypeScript)
typescript

```
import * as aws from "@pulumi/aws";

const org = new aws.organizations.Organization({
   featureSet: "ALL",
});

const devAccount = new aws.organizations.Account("devAccount", {
   name: "DevAccount",
   email: "newaccount@example.com",
});
```

Deploy with Pulumi:

pulumi up

---

3. Service Catalog: Centralized application management.

AWS Service Catalog allows organizations to centrally manage and deploy approved IT services, ensuring compliance and governance. It helps administrators define and control application configurations while enabling self-service provisioning for users.

---

1. AWS UI Configuration

- Navigate to the **AWS Service Catalog** in the AWS Management Console.
- Create a **Portfolio** to organize and manage products.
- Add **Products** (pre-approved AWS resources) using AWS CloudFormation templates.
- Set **Constraints** to enforce limits and configurations.
- Assign **IAM Users, Groups, or Roles** for access control.

---

2. AWS CLI Configuration

aws servicecatalog create-portfolio --name "MyPortfolio" --provider-name "MyCompany"
aws servicecatalog create-product --name "EC2Instance" --owner "Admin" --type "CLOUD_FORMATION_TEMPLATE" --provisioning-artifact-parameters Name="v1",TemplateUrl="https://s3.amazonaws.com/mybucket/template.yaml"
aws servicecatalog associate-product-with-portfolio --product-id <product-id> --portfolio-id <portfolio-id>

---

3. AWS CloudFormation Configuration

yaml

```
Resources:
  MyPortfolio:
    Type: "AWS::ServiceCatalog::Portfolio"
    Properties:
      DisplayName: "MyPortfolio"
      ProviderName: "MyCompany"
```

---

4. Terraform Configuration

hcl

```
resource "aws_servicecatalog_portfolio" "example" {
  name          = "MyPortfolio"
  provider_name = "MyCompany"
}
resource "aws_servicecatalog_product" "example" {
  name          = "EC2Instance"
  owner         = "Admin"
  type          = "CLOUD_FORMATION_TEMPLATE"
  provisioning_artifact_parameters {
    name        = "v1"
    template_url = "https://s3.amazonaws.com/mybucket/template.yaml"
  }
}
```

---

5. Pulumi Configuration (Python)

python

```
import pulumi
import pulumi_aws as aws

portfolio = aws.servicecatalog.Portfolio("myPortfolio",
    name="MyPortfolio",
    provider_name="MyCompany"
)
```

Migration & Transfer

1. **DMS (Database Migration Service)**: Migrate databases to AWS.

Introduction to AWS DMS

AWS **Database Migration Service (DMS)** helps migrate databases to **AWS quickly and securely**. It supports **homogeneous (e.g., MySQL to MySQL)** and **heterogeneous (e.g., Oracle to PostgreSQL)** migrations with minimal downtime.

---

1. AWS DMS Configuration Using AWS UI

1. Open the **AWS Management Console** and navigate to **AWS DMS**.
2. Create a **Replication Instance**:
    - Choose the instance size based on workload.
    - Select the VPC and security group.
3. Create **Source and Target Endpoints**:
    - Provide connection details (database type, credentials, etc.).
4. Create a **Migration Task**:
    - Select the replication instance.
    - Choose migration type (full load, CDC, or both).
    - Map source tables to target.

---

2. AWS DMS Configuration Using AWS CLI

Create Replication Instance

```
aws dms create-replication-instance \
  --replication-instance-identifier my-dms-instance \
  --replication-instance-class dms.t3.medium \
  --allocated-storage 100 \
```

```
  --engine-version 3.4.5 \
  --region us-east-1
```

Create Source Endpoint

```
aws dms create-endpoint \
  --endpoint-identifier source-db \
  --endpoint-type source \
  --engine-name mysql \
  --username admin \
  --password mypassword \
  --server-name my-source-db.cghijk.amazonaws.com \
  --port 3306
```

Create Target Endpoint

```
aws dms create-endpoint \
  --endpoint-identifier target-db \
  --endpoint-type target \
  --engine-name postgres \
  --username admin \
  --password mypassword \
  --server-name my-target-db.cghijk.amazonaws.com \
  --port 5432
```

Create Migration Task

```
aws dms create-replication-task \
  --replication-task-identifier my-migration-task \
  --source-endpoint-arn arn:aws:dms:us-east-1:123456789012:endpoint:source-db \
  --target-endpoint-arn arn:aws:dms:us-east-1:123456789012:endpoint:target-db \
  --migration-type full-load \
  --table-mappings file://table-mappings.json
```

---

3. AWS DMS Configuration Using CloudFormation

Create a CloudFormation template (dms.yaml):

yaml

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  MyDMSInstance:
    Type: AWS::DMS::ReplicationInstance
    Properties:
      ReplicationInstanceIdentifier: "my-dms-instance"
      ReplicationInstanceClass: "dms.t3.medium"
      AllocatedStorage: 100
      EngineVersion: "3.4.5"

  MySourceEndpoint:
    Type: AWS::DMS::Endpoint
    Properties:
      EndpointIdentifier: "source-db"
      EndpointType: "source"
      EngineName: "mysql"
      Username: "admin"
      Password: "mypassword"
      ServerName: "my-source-db.cghijk.amazonaws.com"
      Port: 3306

  MyTargetEndpoint:
    Type: AWS::DMS::Endpoint
    Properties:
      EndpointIdentifier: "target-db"
      EndpointType: "target"
      EngineName: "postgres"
      Username: "admin"
      Password: "mypassword"
```

ServerName: "my-target-db.cghijk.amazonaws.com"
Port: 5432

Deploy with:

aws cloudformation create-stack --stack-name dms-stack --template-body file://dms.yaml

---

4. AWS DMS Configuration Using Terraform

Create a **Terraform file** (dms.tf):

hcl

```
provider "aws" {
  region = "us-east-1"
}

resource "aws_dms_replication_instance" "dms_instance" {
  replication_instance_id = "my-dms-instance"
  replication_instance_class = "dms.t3.medium"
  allocated_storage = 100
  engine_version = "3.4.5"
}

resource "aws_dms_endpoint" "source" {
  endpoint_id = "source-db"
  endpoint_type = "source"
  engine_name  = "mysql"
  username = "admin"
  password = "mypassword"
  server_name = "my-source-db.cghijk.amazonaws.com"
  port = 3306
```

```
}

resource "aws_dms_endpoint" "target" {
  endpoint_id = "target-db"
  endpoint_type = "target"
  engine_name  = "postgres"
  username = "admin"
  password = "mypassword"
  server_name = "my-target-db.cghijk.amazonaws.com"
  port = 5432
}
```

Deploy with:

```
terraform init
terraform apply -auto-approve
```

---

5. AWS DMS Configuration Using Pulumi (Python)

Create a **Pulumi script** (dms.py):

```
import pulumi
import pulumi_aws as aws

# Create DMS Replication Instance
dms_instance = aws.dms.ReplicationInstance("myDMSInstance",
    replication_instance_class="dms.t3.medium",
    allocated_storage=100,
    engine_version="3.4.5",
)

# Create Source Endpoint
source_endpoint = aws.dms.Endpoint("sourceDB",
```

```
    endpoint_type="source",
    engine_name="mysql",
    username="admin",
    password="mypassword",
    server_name="my-source-db.cghijk.amazonaws.com",
    port=3306
)

# Create Target Endpoint
target_endpoint = aws.dms.Endpoint("targetDB",
    endpoint_type="target",
    engine_name="postgres",
    username="admin",
    password="mypassword",
    server_name="my-target-db.cghijk.amazonaws.com",
    port=5432
)

pulumi.export("dms_instance", dms_instance.replication_instance_arn)
```

Deploy with:

```
pulumi up
```

---

2. Migration Hub: Centralized migration tracking.

AWS Migration Hub is a service that helps track the progress of application migrations across multiple AWS and partner tools. It provides a unified dashboard to monitor the status of migrations, whether they are performed using AWS Application Migration Service (MGN), AWS Database Migration Service (DMS), or third-party tools.

Configuration Methods:

1. AWS UI Configuration

- Navigate to AWS Migration Hub.
- Select **Discover** to integrate existing migration tools.
- Use **Migrate** to track progress and link services like AWS MGN or DMS.
- Access the **Progress Dashboard** to monitor migration status.

---

2. AWS CLI Configuration
Enable Migration Hub in your region:

aws migration-hub-config get-home-region

Start tracking a migration:

aws migrationhub import-migration-task --migration-task-name "my-migration" --progress-update-stream "my-stream"

List migration tasks:

aws migrationhub list-migration-tasks

---

3. AWS CloudFormation Configuration

Define a CloudFormation template to enable Migration Hub tracking:

yaml

```
Resources:
  MyMigrationHub:
```

```
  Type: AWS::MigrationHub::ProgressUpdateStream
  Properties:
    ProgressUpdateStreamName: "MyMigrationStream"
```

Deploy the stack:

```
aws cloudformation create-stack --stack-name MigrationHubStack --template-body file://migration-hub.yaml
```

---

4. Terraform Configuration

```
hcl

provider "aws" {
  region = "us-east-1"
}

resource "aws_migrationhub_progress_update_stream" "example" {
  progress_update_stream_name = "my-migration-stream"
}
```

Deploy using:

```
terraform init
terraform apply
```

---

5. Pulumi Configuration (Python Example)

```
import pulumi
import pulumi_aws as aws
```

```
migration_stream =
aws.migrationhub.ProgressUpdateStream("myMigrationStream")

pulumi.export("streamName", migration_stream.progress_update_stream_name)
```

Deploy using:

```
pulumi up
```

---

2. Snowball/Snowcone/Snowmobile: Physical data transfer.

Introduction

AWS **Snow Family (Snowball, Snowcone, Snowmobile)** is a set of **physical data transfer devices** used to migrate large amounts of data to AWS.

- **Snowball**: 80 TB or 100 TB storage appliance for petabyte-scale data transfers.
- **Snowcone**: Small, rugged, and portable (8 TB SSD or 14 TB HDD).
- **Snowmobile**: Exabyte-scale data transfer using a **truck-sized storage container**.

Configuration Methods

1. AWS UI (Console)

1. Go to the **AWS Snow Family Console**.
2. Click **Create Job**.
3. Select **Snowball Edge Compute Optimized, Snowcone, or Snowmobile**.

4. Provide **job details**, including storage capacity and AWS S3 bucket.
5. Confirm shipping details and submit the request.

---

2. AWS CLI

Create a Snowball job:

```
aws snowball create-job \
   --job-type IMPORT \
   --resources S3BucketArn=arn:aws:s3:::your-bucket \
   --address-id <address-id> \
   --snowball-type EDGE
```

List jobs:

```
aws snowball list-jobs
```

---

3. AWS CloudFormation

```
yaml
Resources:
  MySnowballJob:
    Type: AWS::Snowball::Job
    Properties:
      JobType: "IMPORT"
      Resources:
        S3Resources:
          - BucketArn: "arn:aws:s3:::your-bucket"
      SnowballType: "EDGE"
      AddressId: "your-address-id"
```

---

4. Terraform

hcl

```
resource "aws_snowball_job" "example" {
  job_type     = "IMPORT"
  snowball_type = "EDGE"
  address_id   = "your-address-id"

  s3_bucket {
    bucket_arn = "arn:aws:s3:::your-bucket"
  }
}
```

---

5. Pulumi (Python)

python

```
import pulumi
import pulumi_aws as aws

snowball_job = aws.snowball.Job("mySnowballJob",
    job_type="IMPORT",
    snowball_type="EDGE",
    resources=[aws.snowball.JobResourceArgs(
        s3_resources=[aws.snowball.JobResourceS3ResourceArgs(
            bucket_arn="arn:aws:s3:::your-bucket",
        )],
    )],
    address_id="your-address-id"
)
```

---

3. DataSync: Data transfer automation.

AWS DataSync: Data Transfer Automation

AWS DataSync is a managed service that automates and accelerates data transfer between on-premises storage, AWS services, and even between AWS regions. It simplifies large-scale data migrations, backup operations, and continuous data synchronization with security and reliability.

Configuration Methods

1. AWS Management Console (UI)

- Open **AWS DataSync** in the AWS Console.
- Click **Create Task** and configure:
  - **Source Location** (e.g., NFS, SMB, S3, EFS, FSx)
  - **Destination Location** (e.g., S3, EFS, FSx)
- Set transfer options like schedule, filters, and bandwidth limits.
- Start the task and monitor the transfer process.

---

2. AWS CLI Configuration

 # Create a DataSync agent
aws datasync create-agent --activation-key <ACTIVATION_KEY>

 # Create a source location (e.g., NFS)
aws datasync create-location-nfs --server-hostname <NFS_SERVER>
--on-prem-config AgentArns=<AGENT_ARN>

 # Create a destination location (e.g., S3)
aws datasync create-location-s3 --s3-bucket-arn <S3_BUCKET_ARN> --s3-config
BucketAccessRoleArn=<IAM_ROLE_ARN>

 # Create a DataSync task
aws datasync create-task --source-location-arn <SOURCE_ARN>
--destination-location-arn <DESTINATION_ARN> --name "MyDataSyncTask"

 # Start the task
aws datasync start-task-execution --task-arn <TASK_ARN>

---

3. AWS CloudFormation Configuration

Create a datasync.yaml template:

yaml

```
Resources:
  DataSyncTask:
    Type: AWS::DataSync::Task
    Properties:
      SourceLocationArn: !Ref SourceLocation
      DestinationLocationArn: !Ref DestinationLocation
      Name: "MyDataSyncTask"
  SourceLocation:
    Type: AWS::DataSync::LocationNFS
    Properties:
      ServerHostname: "nfs.example.com"
      OnPremConfig:
        AgentArns:
          - "arn:aws:datasync:region:account:agent/agent-id"
  DestinationLocation:
    Type: AWS::DataSync::LocationS3
    Properties:
      S3BucketArn: "arn:aws:s3:::my-bucket"
      S3Config:
```

BucketAccessRoleArn: "arn:aws:iam::account-id:role/MyDataSyncRole"

Deploy using:

aws cloudformation create-stack --stack-name datasync-stack --template-body file://datasync.yaml

---

4. Terraform Configuration

datasync.tf:

hcl

```
resource "aws_datasync_agent" "example" {
  activation_key = "<ACTIVATION_KEY>"
}

resource "aws_datasync_location_nfs" "source" {
  server_hostname = "nfs.example.com"
  on_prem_config {
    agent_arns = [aws_datasync_agent.example.arn]
  }
}

resource "aws_datasync_location_s3" "destination" {
  s3_bucket_arn = "arn:aws:s3:::my-bucket"
  s3_config {
    bucket_access_role_arn = "arn:aws:iam::account-id:role/MyDataSyncRole"
  }
}

resource "aws_datasync_task" "task" {
  source_location_arn = aws_datasync_location_nfs.source.arn
```

```
  destination_location_arn = aws_datasync_location_s3.destination.arn
  name = "MyDataSyncTask"
}
```

Deploy using:

```
terraform init
terraform apply -auto-approve
```

---

5. Pulumi Configuration (Python)

```
python
import pulumi
import pulumi_aws as aws

agent = aws.datasync.Agent("example",
activation_key="<ACTIVATION_KEY>")

source_location = aws.datasync.LocationNfs("source",
    server_hostname="nfs.example.com",
    on_prem_config={"agent_arns": [agent.arn]})

destination_location = aws.datasync.LocationS3("destination",
    s3_bucket_arn="arn:aws:s3:::my-bucket",
    s3_config={"bucket_access_role_arn":
"arn:aws:iam::account-id:role/MyDataSyncRole"})

task = aws.datasync.Task("task",
    source_location_arn=source_location.arn,
    destination_location_arn=destination_location.arn,
    name="MyDataSyncTask")

pulumi.export("task_arn", task.arn)
```

Deploy using:

pulumi up

Media Services

1. MediaConvert: File-based video processing.

Introduction to AWS MediaConvert:

AWS MediaConvert is a file-based video processing service that allows users to convert video files into different formats for broadcasting, streaming, and viewing across a wide range of devices. It provides high-quality video transcoding for multiple media formats.

---

1. Configuration with AWS UI:

1. **Log into the AWS Management Console** and go to the **MediaConvert** service.
2. **Create a new Job**: Click on the "Create job" button and configure the following:
    - **Input**: Select the video file you want to process.
    - **Output**: Choose the desired output format and settings.
    - **Job Settings**: Configure additional options such as resolution, bitrate, and codecs.
3. **Submit the Job**: After configuring, submit the job to start the transcoding process.

---

2. Configuration with AWS CLI:

To submit a MediaConvert job using the AWS CLI:

Install and configure the AWS CLI.

Create a job template or use an existing one:
aws mediaconvert create-job --role arn:aws:iam::account-id:role/role-name --settings file://settings.json

settings.json: This file contains job-specific settings like input files and output settings.

---

3. Configuration with CloudFormation:

CloudFormation allows you to define resources as code. Here's how to configure MediaConvert:

yaml

```
Resources:
  MediaConvertJob:
    Type: AWS::MediaConvert::Job
    Properties:
      Role: arn:aws:iam::account-id:role/role-name
      Settings:
        Input:
          FileInput: "s3://input-bucket/video.mp4"
        OutputGroups:
          - OutputGroupSettings:
              Type: HLS
              HlsGroupSettings:
                Destination: "s3://output-bucket/"
            Outputs:
              - ContainerSettings:
```

```
      Container: M3U8
    VideoDescription:
      Width: 1920
      Height: 1080
      CodecSettings:
        Codec: H_264
```

---

4. Configuration with Terraform:

To create a MediaConvert job using Terraform:

hcl

```
resource "aws_media_convert_job" "example" {
  role = "arn:aws:iam::account-id:role/role-name"

  settings {
    input {
      file_input = "s3://input-bucket/video.mp4"
    }

    output_groups {
      output_group_settings {
        type = "HLS"

        hls_group_settings {
          destination = "s3://output-bucket/"
        }
      }

      outputs {
        container_settings {
          container = "M3U8"
```

```
      }

      video_description {
        width  = 1920
        height = 1080

        codec_settings {
          codec = "H_264"
        }
      }
    }
  }
}
```

---

5. Configuration with Pulumi (Using TypeScript):

Here's how you can configure AWS MediaConvert with Pulumi:

typescript

```
import * as aws from "@pulumi/aws";

const job = new aws.mediaconvert.Job("exampleJob", {
    role: "arn:aws:iam::account-id:role/role-name",
    settings: {
        input: {
            fileInput: "s3://input-bucket/video.mp4",
        },
        outputGroups: [
            {
                outputGroupSettings: {
                    type: "HLS",
```

```
            hlsGroupSettings: {
              destination: "s3://output-bucket/",
            },
          },
          outputs: [
            {
              containerSettings: {
                container: "M3U8",
              },
              videoDescription: {
                width: 1920,
                height: 1080,
                codecSettings: {
                  codec: "H_264",
                },
              },
            },
          ],
        },
      ],
    },
});
```

---

2. MediaLive: Live video processing.

Amazon MediaLive is a fully managed live video processing service that allows users to encode, package, and deliver high-quality video streams to a variety of devices. It is commonly used for broadcasting live events, live TV channels, or on-demand video content with low latency and high availability. MediaLive integrates with other AWS services like MediaPackage, S3, and CloudFront for end-to-end video processing and delivery.

Here’s a small introduction along with configurations for various AWS methods:

1. Configuration using AWS Management Console (UI)

1. **Navigate to MediaLive**:
    - Open the AWS Management Console.
    - Search for **MediaLive** and select the service.
2. **Create a Channel**:
    - Click **Create channel** and configure the **Input** (video source) and **Output group** (destination for your stream).
    - Configure additional settings like encoding parameters, audio settings, and captioning.
3. **Create Input**:
    - Click on **Create input**, define input sources (RTMP, HLS, or other protocols), and configure them.
4. **Start the Channel**:
    - Once configured, select the created channel and click **Start** to begin streaming.

---

2. Configuration using AWS CLI

To configure MediaLive using the AWS CLI, you can use the aws medialive commands.

Create Input:
aws medialive create-input --name MyInput --type RTMP_PUSH

Create Channel:
aws medialive create-channel --name MyChannel --role arn:aws:iam::account-id:role/mediapipeline-role --input-specification resolution=HD

Start the Channel:
aws medialive start-channel --channel-id <channel-id>

---

3. Configuration using CloudFormation

In CloudFormation, you can define the MediaLive channel and input in a YAML or JSON template.

yaml

```
Resources:
  MyMediaLiveChannel:
    Type: AWS::MediaLive::Channel
    Properties:
      Name: "MyChannel"
      RoleArn: arn:aws:iam::account-id:role/mediapipeline-role
      InputAttachments:
        - InputId: arn:aws:medialive:region:account-id:input:MyInput
      Destinations:
        - Id: "Destination1"
          MediaPackageSettings:
            - Destination:
                Url: "rtmp://destination-url"
```

---

4. Configuration using Terraform

In Terraform, you can use the aws_media_live_channel resource.

hcl

```
resource "aws_medialive_channel" "example" {
  name        = "MyMediaLiveChannel"
```

```
  role_arn    = "arn:aws:iam::account-id:role/mediapipeline-role"

  input_attachments {
    input_id = "arn:aws:medialive:region:account-id:input:MyInput"
  }

  destinations {
    id = "Destination1"
    media_package_settings {
      destination = "rtmp://destination-url"
    }
  }
}
```

---

5. Configuration using Pulumi

Using Pulumi with AWS, you can define the MediaLive channel in a similar manner.

typescript

```
import * as aws from "@pulumi/aws";

// Create MediaLive Channel
const channel = new aws.medialive.Channel("myChannel", {
    name: "MyChannel",
    roleArn: "arn:aws:iam::account-id:role/mediapipeline-role",
    inputAttachments: [{
        inputId: "arn:aws:medialive:region:account-id:input:MyInput"
    }],
    destinations: [{
        id: "Destination1",
        mediaPackageSettings: [{
```

```
        destination: "rtmp://destination-url"
      }]
   }]
});
```

---

3. MediaPackage: Video packaging and delivery.

AWS MediaPackage Overview:

AWS MediaPackage is a fully managed video packaging and delivery service. It enables secure and scalable delivery of video content to a wide range of devices. MediaPackage supports various protocols like HLS, DASH, CMAF, and Microsoft Smooth Streaming, making it ideal for live streaming and on-demand video workflows.

Configuration of AWS MediaPackage:

1. AWS UI (Management Console):

- **Step 1:** Open the AWS Management Console.
- **Step 2:** Navigate to **MediaPackage** under the "Media Services" section.
- **Step 3:** Click on **Create Channel** to create a new video channel.
- **Step 4:** Set up a **Channel**, **Origin Endpoint**, and specify video format and delivery options.
- **Step 5:** Configure output destinations like S3, and set up **HLS**, **DASH**, or other formats.
- **Step 6:** Review and create the MediaPackage resource.

---

2. AWS CLI:

- **Step 1:** Install the AWS CLI.
- **Step 2:** Use the aws mediapackage command to create resources.

Example command to create a channel:
aws mediapackage create-channel --id my-channel --description "My Video Channel"
Step 3: Use create-origin-endpoint to define the delivery settings:
aws mediapackage create-origin-endpoint --channel-id my-channel --id my-endpoint --manifest-name my-manifest --hls-package "{...}"

---

3. AWS CloudFormation:

CloudFormation allows you to define MediaPackage resources using YAML or JSON templates.

Example CloudFormation template for MediaPackage:
yaml

```
Resources:
  MyChannel:
    Type: AWS::MediaPackage::Channel
    Properties:
      Id: my-channel
      Description: "My Video Channel"

  MyEndpoint:
    Type: AWS::MediaPackage::OriginEndpoint
    Properties:
      ChannelId: !Ref MyChannel
      Id: my-endpoint
      ManifestName: my-manifest
      HlsPackage:
```

```
    SegmentDurationSeconds: 10
    PlaylistType: VOD
```

---

4. Terraform:

With Terraform, you can define MediaPackage resources in .tf files.

Example Terraform configuration:

```
provider "aws" {
  region = "us-west-2"
}

resource "aws_media_package_channel" "my_channel" {
  id          = "my-channel"
  description = "My Video Channel"
}

resource "aws_media_package_origin_endpoint" "my_endpoint" {
  channel_id = aws_media_package_channel.my_channel.id
  id         = "my-endpoint"
  manifest_name = "my-manifest"

  hls_package {
    segment_duration_seconds = 10
    playlist_type            = "VOD"
  }
}
```

---

5. Pulumi:

Pulumi is an infrastructure-as-code tool that supports AWS MediaPackage.

Example Pulumi code:
typescript

```
import * as aws from "@pulumi/aws";

const mediaPackageChannel = new aws.mediapackage.Channel("my-channel", {
  description: "My Video Channel",
});

const mediaPackageEndpoint = new
aws.mediapackage.OriginEndpoint("my-endpoint", {
  channelId: mediaPackageChannel.id,
  manifestName: "my-manifest",
  hlsPackage: {
    segmentDurationSeconds: 10,
    playlistType: "VOD",
  },
});
```