Infrastructure as Code (IaC) Project using Terraform

The goal of this project is to implement **Infrastructure as Code (IaC)** using **Terraform** to provision and manage AWS resources. The project focuses on creating reusable templates for infrastructure deployment and updates, ensuring automation, ease of deployment, and adherence to best practices.

1. Introduction

This project uses **Terraform** to provision AWS resources, enabling consistent and repeatable infrastructure management. The configuration is modular, reusable, and follows IaC principles to automate deployments and ensure scalability.

2. Project Objectives

- Automate infrastructure provisioning using Terraform.
- Create reusable, parameterized templates for AWS resources.
- Enable efficient infrastructure updates and configuration changes.
- Integrate security, monitoring, and logging mechanisms.
- Follow AWS best practices for scalability, high availability, and security.

3. Technology Stack

- **Terraform**: Infrastructure as Code (IaC) tool for managing AWS resources.
- **AWS Services**:
  - **VPC**: Virtual Private Cloud for networking.
  - **Subnets**: Public and private subnets for resource segregation.
  - **EC2**: Compute resources for virtual machines.
  - **Security Groups**: Network security at the instance level.
  - **CloudWatch**: Monitoring and alerting.
  - **IAM**: Access control and permissions.
- **Tools**:
  - AWS CLI for management and automation.
  - Git for version control.

4. Infrastructure Components

Resources Provisioned:

1. **VPC**: A custom VPC with CIDR block `10.0.0.0/16`.
2. **Subnets**:

    - Public Subnet with CIDR `10.0.1.0/24`.
    - Private Subnet with CIDR `10.0.2.0/24`.
3. **Internet Gateway**: Provides internet access to the public subnet.
4. **Route Table**: Configured for public and private routing.
5. **Security Group**:
    - Allows inbound SSH (port 22) and HTTP (port 80).
    - Denies all other traffic.
6. **EC2 Instance**: Amazon Linux 2 instance in the public subnet.
7. **CloudWatch Logs**: Monitors EC2 instance activity.

5. Project Structure

```
terraform-project/
│
├── main.tf          # Main configuration file for resources
├── variables.tf     # Input variables for parameterization
├── outputs.tf       # Outputs for resource details
├── provider.tf      # AWS provider configuration
├── terraform.tfvars  # Variable values (not to be committed to Git)
└── README.md        # Documentation for the project
```

6. Steps to Deploy

Install Terraform: Configure AWS CLI: (aws configure)

Initialize Terraform:

```
terraform init
```

Review Variables

Check the `variables.tf` file for customizable parameters. Update `terraform.tfvars` with your values. Example:

```
vpc_cidr = "10.0.0.0/16"
public_subnet_cidr = "10.0.1.0/24"
private_subnet_cidr = "10.0.2.0/24"
instance_type = “t2.micro"
```

Plan the Infrastructure

```
terraform plan
```

Apply the Configuration

```
terraform apply
```

Verify Deployment

1. Check the AWS Management Console for created resources.
2. Verify the public IP of the EC2 instance is accessible via SSH:`bash`

```
ssh -i <key.pem> ec2-user@<public-ip>
```

Updating Infrastructure

1. Modify any `.tf` file (e.g., change the instance type).
2. Run `terraform plan` to review changes.
3. Apply updates with:`bash`
   `terraform apply`

Best Practices Followed

- **Modular Design**: Configuration uses input variables for reusability.
- **Version Control**: Git is used to track changes.
- **State Management**: Terraform state is stored locally or can be configured for remote storage (e.g., S3).
- **Security**:
  - SSH key pair for EC2 instance access.
  - Minimal IAM permissions for Terraform.
  - Restricted CIDR ranges in security groups.

Future Enhancements

- Add **Auto Scaling** and **Load Balancers** for high availability.
- Enable **Remote State Storage** in S3 with DynamoDB locking.
- Integrate Terraform with **CI/CD pipelines** for automated deployments.
- Include **SNS notifications** for CloudWatch alarms.
- Add support for multi-region deployments.

Conclusion

This project demonstrates the power of Terraform for provisioning and managing AWS resources with an automated, reusable approach. The implementation ensures scalability, security, and ease of management, aligning with IaC principles and AWS best practices.