Infrastructure Provisioning with AWS & CI CD Objectives

Introduction

This guide outlines the steps to provision a basic AWS infrastructure for hosting a web application. The setup includes creating an EC2 instance, configuring security settings to allow HTTP traffic, and establishing a Virtual Private Cloud (VPC) with a public subnet. Additionally, the guide details the integration of a CI/CD pipeline using GitLab to automate the deployment of the application. Automation options using Terraform are also included for efficient and repeatable deployments.

1. Overview

This guide provides instructions to:

- **Set up AWS infrastructure** using Terraform.
- **Build, push, and deploy a Flask application** containerized with Docker to AWS ECS.
- **Test endpoints and validate the deployment.**

---

2. AWS Infrastructure Setup

Prerequisites

- **AWS CLI** installed and configured with sufficient permissions.
- **Terraform CLI** installed.

Steps
Initialize Terraform

```
terraform init
```

Review and Apply Terraform Configurations

  - Verify the resources defined in the `.tf` files.

Apply the configurations:

```
terraform plan
Terraform apply
```

    - This will create:
        - VPC, subnets, and internet gateway (`network.tf`).
        - ECS task execution IAM role and policy (`ecs-iam.tf` and `iam.tf`).
        - An ECR repository for the application (`ecr.tf`).
        - Outputs the ECR repository URL (`outputs.tf`).

Verify Outputs

    - Copy the `ecr_repository_url` from the output. You'll use this URL to push your Docker image.

---

3. Flask Application

Application Details

- The Flask application provides:
    - A health check endpoint: `/health`
    - A root endpoint: `/`

Code Structure

- **`app.py`**: Main Flask app code with logging and endpoints.
- **`test_app.py`**: Pytest-based tests for the Flask application.

---

4. Containerizing the Flask App

Steps

Build the Docker Image Navigate to the `python-app` directory and build the image:
bash
Copy code

```
cd python-app
docker build -t flask-app .
```

---

5. Deploying to AWS ECS

Infrastructure Assumptions

- The ECS cluster, ALB, and related services are created via Terraform (defined in `alb.tf` and other files).

Steps

1. **Create an ECS Task Definition**
   - Use the `ecs-task-execution-role` for execution.
   - Reference the ECR image `<ECR_REPOSITORY_URL>:latest`.
2. **Deploy the ECS Service**
   - Attach the service to the ALB.
   - Ensure the ALB forwards traffic from port `80` to your ECS tasks (on port `5000`).

---

6. Testing the Deployment

Verify Application Endpoints

1. Fetch the ALB's DNS name or public IP (from the AWS console or Terraform output).
2. Test the endpoints using `curl` or a browser:

Root endpoint:

```
curl http://<ALB-DNS-NAME>/
```

Expected output:
json

```
{"message": "Hello, World!"}
```

○

Health check endpoint:

```
curl http://<ALB-DNS-NAME>/health
```

Expected output:
json

```
{
  "status": "healthy",
  "timestamp": "...",
  "version": "1.0.0",
  "python_version": "...",
  "environment": "production"
}
```

Run Tests Locally

Navigate to `python-app` and run the tests:

```
pytest app/test_app.py
```

Expected output:

```
2 passed in 0.XXs
```

---

7. CI/CD Pipeline

Define CI/CD Pipeline in `ci-cd.yaml`

- Automate the following tasks:
    1. Run unit tests.
    2. Build the Docker image.
    3. Push the image to ECR.
    4. Deploy updated tasks to ECS.

8. Key Notes

- **Security Best Practices:**
    - Restrict IAM roles and security group rules to the least privilege.
    - Use HTTPS for ALB listeners with an ACM certificate.
- **Scalability:**
    - Update ECS service configurations for auto-scaling based on CPU/memory usage.
- **Monitoring:**
    - Enable CloudWatch logging for ECS tasks.
    - Set up ALB access logs for detailed insights.