Fibonacci Python App Deployment on EKS Using Terraform

This document outlines the design and deployment of a Fibonacci Python app on Amazon EKS (Elastic Kubernetes Service) with load balancing capabilities, automated deployment using Terraform, and instructions for deployment, updates, and testing.

Assumptions

1. The AWS CLI is installed and configured with necessary IAM permissions.
2. Terraform is installed on the local machine.
3. kubectl is installed for Kubernetes command-line access.
4. Helm is installed for managing Kubernetes applications.
5. Docker CLI is installed

Design

1. **EKS:** Kubernetes as a Service from AWS. It is highly available, scalable, and has deep integrations with AWS services.
2. **Terraform:** IAC - Maintains the state of the infrastructure, necessary for maintaining and updating the resources.
3. **Github Actions CI/CD:** We can release our code to terraform using a pipeline built through Github Actions, building/releasing our infrastructure to different stages.
4. **AWS Elastic Load Balancer (ELB):** Distributes incoming traffic across multiple nodes on our kube cluster.
5. **AWS Auto Scaling:** To ensure high availability, scalability, and cost efficiency.
6. **AWS S3:** Used to store our Terraform state files. S3 provides a reliable and secure solution for storing these files.
7. **AWS ECR:** Used to store our container builds.
8. **AWS Dynamo:** Used for Terraform state - locking etc.

Deployment Process

The deployment will be executed through Terraform, as detailed by the structure of the Terraform files:

1. `providers.tf`: Holds information about the AWS provider.
2. `vpc.tf`: Defines the configuration for VPC, Subnet, Route, and Internet Gateway.
3. `eks.tf`: Establishes the EKS cluster configuration.
4. `elb.tf`: Specifies the configuration for the Application Load Balancer.
5. `data.tf`: Contains data definitions used across other files.
6. `ecr.tf`: Contains the Elastic Container Registry configuration.
7. `locals.tf`: Contains local variable definitions.
8. `outputs.tf`: Holds the output definitions.
9. `variables.tf`: Provides variable definitions used across multiple configuration files.
10. `terraform.tfstate` and `terraform.tfstate.backup`: Contain the Terraform state and its backup respectively.

The deployment also includes additional files and directories:

- `LICENSE`: The license file for this codebase.
- `README.md`: The file providing details about the codebase and its usage.
- `config`: This directory contains `dev.tfvars`, a file for variable definitions specific to the development environment.
- `eks-learnings.code-workspace`: Workspace file for this project.
- `manifests`: This directory contains `deployment.yaml` and `elb.yaml` files, defining Kubernetes resources.

Instructions for Deployment, Updates, and Testing

Deployment Infra:

1. Clone the repository containing the Terraform files.
2. Run `terraform init` to initialize your Terraform workspace.
3. Run `terraform plan -var-file=config/dev.tfvars` to see what changes will be applied.
4. Run `terraform apply -var-file=config/dev.tfvars` to apply the changes.

Deployment Code:

1. `docker buildx build --platform linux/amd64 -t {AWS ECR URL}/{NAME}:{TAG} --push .` - For M1 Mac building amd64 and arm image for hosting on ECR
2. `kubectl apply -f elb.yaml - Deploy load balancer`
3. `kubectl apply -f deployment.yaml - Deploy application`

Updates:

1. Make necessary changes in the appropriate `.tf` files.
2. Run `terraform plan -var-file=config/dev.tfvars` to see the changes.
3. Run `terraform apply -var-file=config/dev.tfvars` to apply the changes.

Testing:

1. Run `kubectl get nodes` to ensure your EKS cluster is running.
2. Run `kubectl get pods -n {namespace}` to ensure your pods are running.
3. Use the AWS ELB URL to access the application and ensure it's working.
4. Use a load testing tool like JMeter or Gatling to test the load balancing and auto-scaling functionalities.

Problems

- Currently the app is run in a stateless way. Each container is keeping track of it's own fibonacci number, so depending on which server is queried, an unexpected result may be returned. This could be fixed by making use of a database to store/retrieve the latest fibonacci number

Possible Extensions

- Make use of AWS RDS to store fibonacci number in and extend application to retrieve the last fibonacci number from there and update with the latest following being run
- Different users could have different session keys as well in order to retrieve their own set of fibonacci numbers rather than receiving someone elses

- Extension using user authentication/SSO for login to get session key / userid for retrieval of last generated number from database
- Django seems like an unnecessary choice for a lightweight api, rewrite to use fastapi/flask or switch for an amazon primitive such as API Gateway
- Switch out AWS ELB for a better solution - My test environment could not accommodate NLB/ALB. Ingress Controller with Nginx via helm or something
- Build out CI/CD pipeline using github actions for container build and release, implementing testing. A rolling deployment would be suitable for this task.
- Migrate state to use remote state on s3/dynamo.
- Separate into dev/production environments
- Use tf workspaces to separate dev and prod workspaces