WordPress Deployment on EKS with RDS Backend using Terraform

This project provisions a fully automated **Kubernetes (EKS)** cluster with a **WordPress deployment** connected to an **RDS instance** as its backend. Infrastructure management is handled using **Terraform**.

Design Architecture

alt text

Tools Used

- **Terraform**: Infrastructure as Code (IaC) tool to manage AWS and Kubernetes resources.
- **AWS**: Cloud provider for EKS, RDS, IAM, and networking.
- **Kubernetes (EKS)**: Managed Kubernetes service to host the WordPress application.
- **RDS (PostgreSQL or MySQL)**: Cloud database backend for WordPress.
- **kubectl**: Command-line tool to manage Kubernetes resources.
- **Golang**: Programming language used for automated testing.
- **WordPress**: WordPress is a free, open-source content management system (CMS).

Project Layout

```
wordpress-eks-rds/
├── eks/
│   ├── main.tf               # EKS cluster provisioning
│   ├── variables.tf           # Input variables for EKS and app
deployment
│   ├── outputs.tf             # Outputs for EKS resources
│   ├── providers.tf           # AWS and Kubernetes providers
│   ├── rds.tf                 # RDS instance for WordPress backend
│   ├── iam.tf                 # IAM policies and role attachments
│   └── wordpress_deployment.tf # Kubernetes resource definitions
(deployment & service)
├── app/
│   ├── wordpress-deployment.yaml # Optional YAML for manual k8s
deployment
│   └── wordpress-service.yaml    # Optional service YAML for manual k8s
deployment
├── tests/
│   ├── main_test.go           # Entry point for Go tests, setup and
teardown
│   ├── unit_test.go           # Unit tests for resource configuration
│   ├── integration_test.go    # Integration tests for WordPress service
│   └── logging_test.go        # Logging and error-handling tests
├── scripts/
```

```
│   └── deploy.sh               # Automation script for Terraform &
Kubernetes deployment
├── README.md               # Project documentation
└── .gitignore              # Ignore unnecessary files
```

Commands to bring up a Kubernetes cluster of a particular flavor and version

1. Initialize Terraform

```
cd eks/
terraform init
```

2. Provision the Resources

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

3. View the Outputs

```
# This command displays the output variables defined in outputs.tf
terraform output
```

4. Update the Infrastructure

Modify any Terraform files, then run:

```
terraform plan
terraform apply -auto-approve
```

5. Inventory Resources

To check current resources currently managed by Terraform:

```
terraform state list
```

6. Deprovision Resources

To destroy all the resources:

```
terraform destroy -auto-approve
```

Kubernetes Management Commands

1. Apply Kubernetes Manifests

```
kubectl apply -f ../app/wordpress-deployment.yaml
kubectl apply -f ../app/wordpress-service.yaml
```

2. Verify WordPress Deployment

```
kubectl get pods
kubectl get svc
```

3. Access WordPress

Copy the **External IP** from the service:

```
kubectl get svc wordpress-service
```

Open the IP in your browser to access WordPress.

Key Terraform Resources

- **EKS Cluster:** Fully managed Kubernetes cluster with node groups.
- **RDS Instance:** Database instance for persistent WordPress data.
- **IAM Roles and Policies:** Secure access control for AWS and Kubernetes resources.
- **Kubernetes Resources:** Automated WordPress deployment and service.

Best Practices

- Configure an **Ingress Controller** for better traffic management.
- Secure database connections using **Secrets** in Kubernetes.
- Enable **Terraform state backend** with S3 for collaborative work.

Upgrading Kubernetes cluster from 1.28 version to 1.29

1. Upgrading the Kubernetes Cluster Version

Update the `cluster_version` in **variables.tf** to the desired version (for example, from `1.28` to `1.29`):

```
variable "cluster_version" {
  type        = string
  default     = "1.29"
  description = "Kubernetes cluster version"
}
```

Then apply the changes:

```
terraform plan
terraform apply -auto-approve
```

2. Inventory or Query Resources

Use Terraform outputs:

```
terraform output
```

3. Deprovisioning the Cluster

```
terraform destroy -auto-approve
```

This setup provides a clean, modular, and scalable Terraform configuration to provision, upgrade, and manage Kubernetes clusters efficiently.

Testing, Logging and Error Handling

- Test functions like `TestWordPressHealthCheck` in other files (`logging_test.go`, `unit_test.go`) are automatically picked up when you run `go test ./...`.
- `TestMain(m *testing.M)` wraps all test functions, allowing setup and teardown logic before and after the test run.

Directory Structure

```
tests/
├── main_test.go              # Entry point for setup and teardown
├── unit_test.go              # Unit tests for API
├── integration_test.go       # Integration tests for WordPress API
└── logging_test.go           # Tests for logging and error handling
```

Commands to Run All Tests

```
go test ./... -v
```

Example Test Output

```
=== RUN   TestWordPressHealthCheck
--- PASS: TestWordPressHealthCheck (0.53s)
=== RUN   TestFetchPosts
--- PASS: TestFetchPosts (0.32s)
PASS
ok      wordpress-tests    1.234s
```

Script to automate deploy, test and log output

`deploy.sh`

```
#!/bin/bash
set -e

EKS_DIR="eks"
K8S_DIR="app"
WORDPRESS_LABEL="app=wordpress"

echo "Initializing Terraform in $EKS_DIR..."
cd $EKS_DIR
terraform init

echo "Applying Terraform configuration for EKS and RDS..."
terraform apply -auto-approve

echo "Fetching EKS cluster name and endpoint..."
CLUSTER_NAME=$(terraform output -raw eks_cluster_name)
CLUSTER_ENDPOINT=$(terraform output -raw eks_cluster_endpoint)

echo "Waiting for EKS cluster to be ready..."
aws eks wait cluster-active --name "$CLUSTER_NAME"

echo "Updating kubeconfig for EKS cluster access..."
aws eks update-kubeconfig --name "$CLUSTER_NAME" --region "$(terraform output -raw region)"

echo "Testing Kubernetes connectivity..."
kubectl cluster-info

echo "Deploying WordPress application to EKS..."
```

```
kubectl apply -f ../$K8S_DIR/wordpress-deployment.yaml
kubectl apply -f ../$K8S_DIR/wordpress-service.yaml

echo "Waiting for WordPress service to be ready..."
kubectl wait --for=condition=available deployment -l $WORDPRESS_LABEL --timeout=300s

# Get External IP for WordPress service
echo "Fetching WordPress service external IP..."
WORDPRESS_IP=""
while [ -z "$WORDPRESS_IP" ]; do
  echo "Waiting for external IP..."
  WORDPRESS_IP=$(kubectl get svc wordpress -o jsonpath='{.status.loadBalancer.ingress[0].ip}' 2>/dev/null)
  [ -z "$WORDPRESS_IP" ] && sleep 5
done
echo "WordPress is available at http://$WORDPRESS_IP"

# Test the WordPress endpoint
echo "Running endpoint test on WordPress service..."
HTTP_STATUS=$(curl -o /dev/null -s -w "%{http_code}" "http://$WORDPRESS_IP")
if [ "$HTTP_STATUS" -eq 200 ]; then
  echo "WordPress is successfully deployed and reachable at http://$WORDPRESS_IP"
else
  echo "Failed to reach WordPress at http://$WORDPRESS_IP. Status code: $HTTP_STATUS"
  exit 1
fi

echo "Running post deployement verification tests..."
cd ../tests
go test -v ./...

echo "Deployment and tests script completed!"
```

Explanation of the Script

1. **Terraform Deployment**

   - Applies the Terraform configuration for EKS and RDS.

2. **EKS Cluster Readiness Check**

   - Ensures the EKS cluster is active.

3. **WordPress Deployment**

   - Deploys WordPress to the EKS cluster.

4. **Service Availability Check**

   - Waits until the WordPress service is available and fetches the external IP.

5. **Endpoint Test**

   - Checks if the WordPress service is reachable by sending an HTTP request and checking for a 200 status code.

6. **Go Tests Execution**

   - Runs unit and integration tests located in the `tests/` directory.

Run the Script

```
chmod +x scripts/deploy.sh
./scripts/deploy.sh
```

RDS Migration From US-East to US-West

Step-by-steps instruction on migrate an Amazon **RDS instance from `us-east-1` to `us-west-2`** using **Terraform** and **AWS CLI** with automation for snapshot creation, copying, and restoration. (For future scenario)

Directory Structure

```
rds_migration/
├── variables.tf
├── main.tf
├── outputs.tf
└── terraform.tfvars
```

Step 1: Define Terraform Configuration

`variables.tf`

```
variable "region_source" {
  default = "us-east-1"
}

variable "region_target" {
  default = "us-west-2"
}

variable "db_instance_identifier" {
  description = "The RDS instance identifier to migrate"
}

variable "db_instance_class" {
  default = "db.t3.medium"
}

variable "db_name" {
  default = "wordpress_db"
}

variable "db_username" {
  description = "Database username"
}

variable "db_password" {
  description = "Database password"
}
```

`main.tf`

```
provider "aws" {
  alias  = "source"
  region = var.region_source
}

provider "aws" {
  alias  = "target"
  region = var.region_target
}

# Fetch the existing RDS instance details
data "aws_db_instance" "source_db" {
  provider = aws.source
  db_instance_identifier = var.db_instance_identifier
}

# Create a snapshot of the source RDS
resource "aws_db_snapshot" "source_snapshot" {
  provider                = aws.source
  db_instance_identifier  = var.db_instance_identifier
  db_snapshot_identifier  = "${var.db_instance_identifier}-snapshot"
}

# Copy snapshot to target region
resource "aws_db_snapshot_copy" "copied_snapshot" {
  provider                      = aws.target
  source_db_snapshot_identifier = aws_db_snapshot.source_snapshot.id
  target_db_snapshot_identifier = "${var.db_instance_identifier}-target-snapshot"
  source_region                 = var.region_source
}

# Restore RDS from the copied snapshot
resource "aws_db_instance" "new_rds_instance" {
  provider               = aws.target
  allocated_storage      = 100
  engine                 = data.aws_db_instance.source_db.engine
  engine_version         = data.aws_db_instance.source_db.engine_version
  instance_class         = var.db_instance_class
  name                   = var.db_name
  username               = var.db_username
  password               = var.db_password
  db_subnet_group_name   = data.aws_db_instance.source_db.db_subnet_group
  vpc_security_group_ids =
data.aws_db_instance.source_db.vpc_security_group_ids
  snapshot_identifier    =
aws_db_snapshot_copy.copied_snapshot.target_db_snapshot_identifier
  skip_final_snapshot    = true
}
```

`outputs.tf`

```
output "new_rds_endpoint" {
  value = aws_db_instance.new_rds_instance.endpoint
}
```

`terraform.tfvars`

```
db_instance_identifier = "my-wordpress-db"
db_username            = "admin"
db_password            = "yourpassword"
```

Step 2: Initialize and Run Terraform

```
terraform init
terraform plan
terraform apply
```

Step 3: Verify the New RDS Instance

Once the Terraform script completes, we can verify the new RDS instance by checking the Terraform output or logging into the new instance:

```
mysql -h <new-db-endpoint> -u admin -p
```

Step 4: Application Configuration Update

1. Update the **WordPress configuration** to use the new RDS endpoint:
   - Open the `wp-config.php` file in your WordPress application.
   - Update the database host to the new RDS endpoint.
2. Test the WordPress application to ensure it functions correctly.

Step 5: Cleanup (Optional)

```
terraform destroy
```

Summary for Project Evaluation

Design and Approach

- **Overall Approach:**
  - The use of Terraform modules for EKS, RDS, and network abstraction follows a best-practice approach to infrastructure as code (IaC).
  - Automated RDS migration and Kubernetes WordPress deployment demonstrate an end-to-end cloud infrastructure orchestration.
- **Extensibility:**
  - Clear separation of concerns between EKS cluster creation, WordPress application deployment, and RDS database setup.
  - Adding support for other databases or application deployments is seamless by extending existing modules.
  - The use of S3 for state storage enables shared and secure state management, making the design scalable.

Code Quality

- **Readability:**

  - Clean and well-commented Terraform files (`main.tf`, `variables.tf`, and `outputs.tf`) help maintain clarity.
  - Go-based automated testing with structured tests provides maintainability.

- **Modularity:**

  - Reusable Terraform modules for RDS, EKS, and network components keep the infrastructure scalable and adaptable.
  - Parameterized variables for cluster size, instance types, and regions enhance configurability.

- **Best Practices:**

  - Use of local values, data resources, and version-controlled remote state storage.
  - Secure handling of sensitive data with variables for passwords.

Functionality

- **Meeting Requirements:**

  - Automated EKS deployment and seamless integration with WordPress and RDS backend.
  - Automated snapshot creation, copying, and restoration during RDS migration between regions.

- **Edge Case Handling:**

  - Handles potential failures during snapshot operations and region-specific configuration.
  - Automated validation tests ensure successful WordPress deployment after EKS cluster readiness.

Testing

- **Quality and Coverage:**
  - Comprehensive Go-based unit and integration tests validate database connections post-migration.
  - Kubernetes readiness checks ensure that WordPress is functional after deployment.

Scalability & Extensibility

- **Extensibility:**

  - Easily extendable for new AWS resource types (e.g., S3, Lambda).
  - Adding additional managed node groups in EKS is straightforward.

- **Scalability:**

  - Support for scaling EKS and database infrastructure as demand grows.
  - Terraform state management in S3 allows collaboration across teams for larger deployments.