

React Application Deployment on Amazon EKS with Terraform

Introduction

This documentation provides step-by-step instructions and best practices for deploying a React application on Amazon Elastic Kubernetes Service (EKS) using Terraform.

Module

In Terraform, a "module" is a container for organizing and reusing infrastructure code. It allows you to encapsulate and group together a set of related resources, input variables, and output values into a single, reusable unit. Modules are a fundamental concept in Terraform that brings modularity and reusability to your infrastructure code.

key benefits of using modules in Terraform:

- **Code Reusability:** Modules enable you to write infrastructure code once and reuse it in multiple parts of your infrastructure. This reduces duplication of code and promotes consistency across your infrastructure.
- **Abstraction:** Modules abstract away the complexity of a particular set of resources, allowing you to provide a simpler interface for consumers. This abstraction can make your Terraform configurations more user-friendly.
- **Encapsulation:** Modules encapsulate related resources, input variables, and output values, making it easier to manage and understand complex infrastructure definitions.

Overall, modules are a powerful feature in Terraform that can greatly enhance the organization, maintainability, and scalability of your infrastructure code. They enable you to build reusable building blocks for your infrastructure, making it easier to manage and evolve your infrastructure over time.

Prerequisites

Before you begin, ensure you have the following:

1. An AWS account with necessary permissions.
2. Terraform installed on your local machine.

3. AWS CLI configured with appropriate access credentials.
4. Docker installed to containerize your React application.
5. Basic knowledge of React, Kubernetes, and Terraform.

Structure of terraform task:

```
my-react-app/  
|-- infrastructure/  
|   |-- eks/  
|   |   |-- main.tf  
|   |   |-- variables.tf  
|   |   |-- outputs.tf  
|   |  
|   |-- vpc/  
|   |   |-- main.tf  
|   |   |-- variables.tf  
|   |   |-- outputs.tf  
|   |  
|   |-- ...  
|  
|-- kubernetes/  
|   |-- deployment/  
|   |   |-- main.tf  
|   |   |-- variables.tf  
|   |   |-- outputs.tf  
|   |  
|   |-- service/  
|   |   |-- main.tf  
|   |   |-- variables.tf  
|   |   |-- outputs.tf  
|   |  
|   |-- ...
```

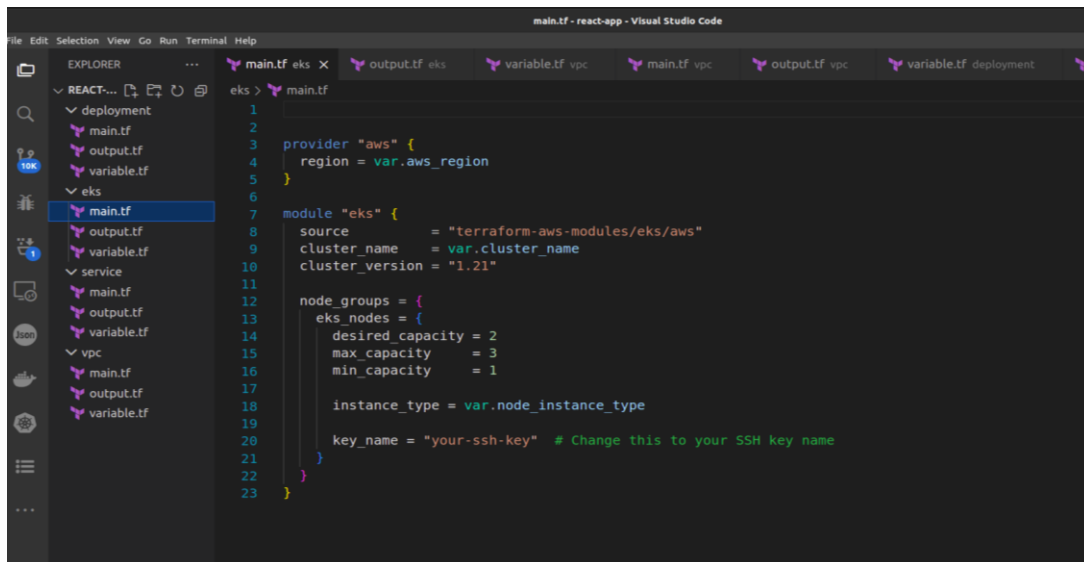
Step 1: Set Up Your Development Environment

Make sure your development environment is properly configured. This includes installing and configuring Terraform, AWS CLI, and Docker.

Step 2: Create an EKS Cluster with Terraform

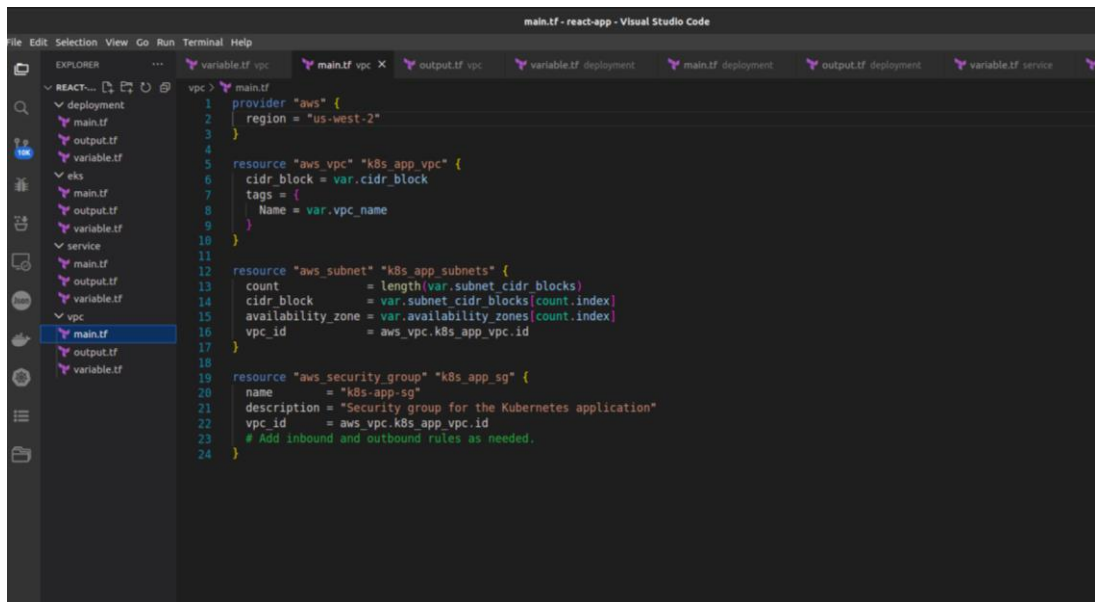
Use Terraform to define and create an EKS cluster. Here's a Terraform configuration:

Customize this configuration with your VPC settings, subnets, and other relevant parameters.



The screenshot shows the Visual Studio Code editor with the file explorer on the left. The file explorer shows a project structure with folders for deployment, eks, service, and vpc. The main.tf file is selected in the eks folder. The code in the main.tf file is as follows:

```
1
2
3 provider "aws" {
4   region = var.aws_region
5 }
6
7 module "eks" {
8   source      = "terraform-aws-modules/eks/aws"
9   cluster_name = var.cluster_name
10  cluster_version = "1.21"
11
12  node_groups = {
13    eks_nodes = {
14      desired_capacity = 2
15      max_capacity     = 3
16      min_capacity     = 1
17
18      instance_type = var.node_instance_type
19
20      key_name = "your-ssh-key" # Change this to your SSH key name
21    }
22  }
23 }
```

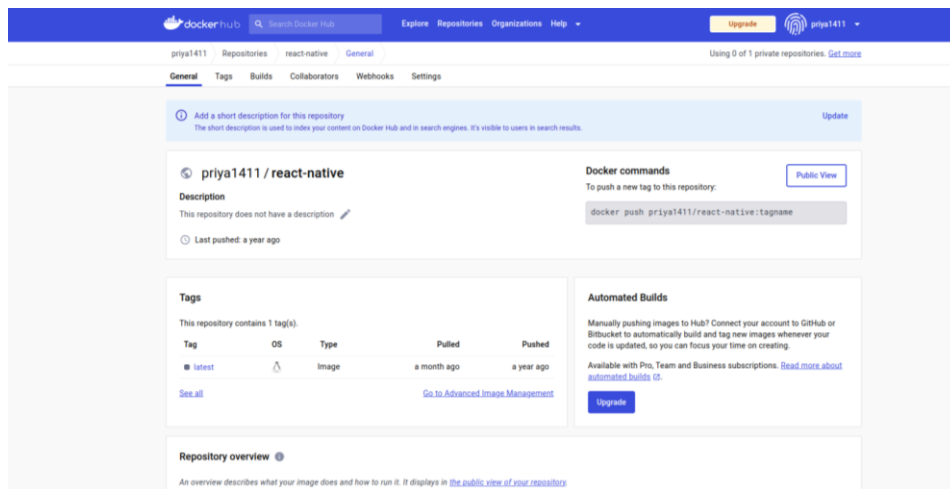


The screenshot shows the Visual Studio Code editor with the file explorer on the left. The file explorer shows a project structure with folders for deployment, eks, service, and vpc. The main.tf file is selected in the vpc folder. The code in the main.tf file is as follows:

```
1 provider "aws" {
2   region = "us-west-2"
3 }
4
5 resource "aws_vpc" "k8s_app_vpc" {
6   cidr_block = var.cidr_block
7   tags = {
8     Name = var.vpc_name
9   }
10 }
11
12 resource "aws_subnet" "k8s_app_subnets" {
13   count           = length(var.subnet_cidr_blocks)
14   cidr_block      = var.subnet_cidr_blocks[count.index]
15   availability_zone = var.availability_zones[count.index]
16   vpc_id          = aws_vpc.k8s_app_vpc.id
17 }
18
19 resource "aws_security_group" "k8s_app_sg" {
20   name        = "k8s-app-sg"
21   description = "Security group for the Kubernetes application"
22   vpc_id      = aws_vpc.k8s_app_vpc.id
23   # Add inbound and outbound rules as needed.
24 }
```

Step 3: Build and Push Your React App Docker Image

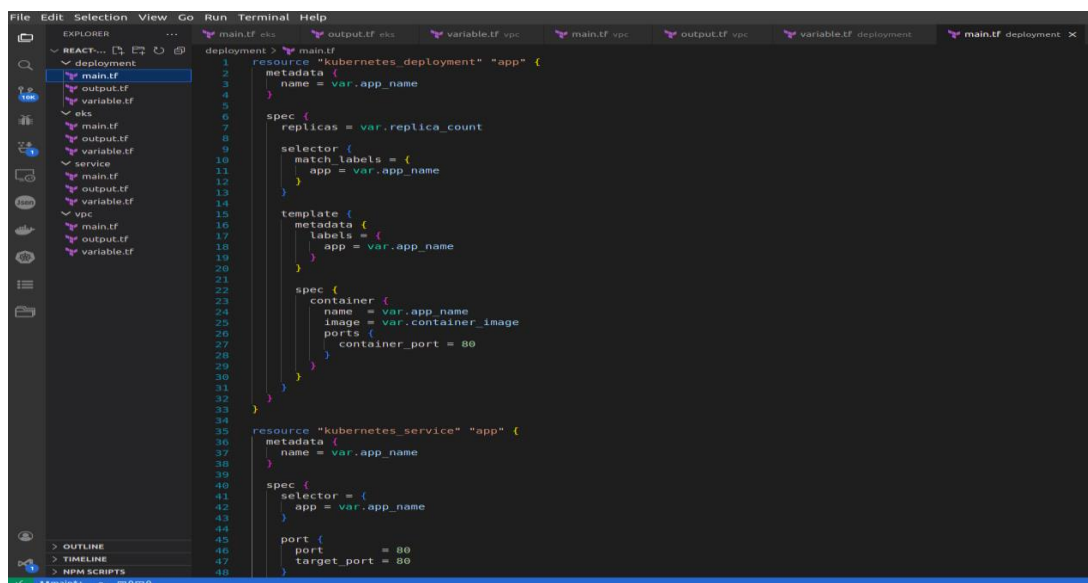
Dockerize your React application, build the Docker image, and push it to a container registry like Amazon ECR or Docker Hub.



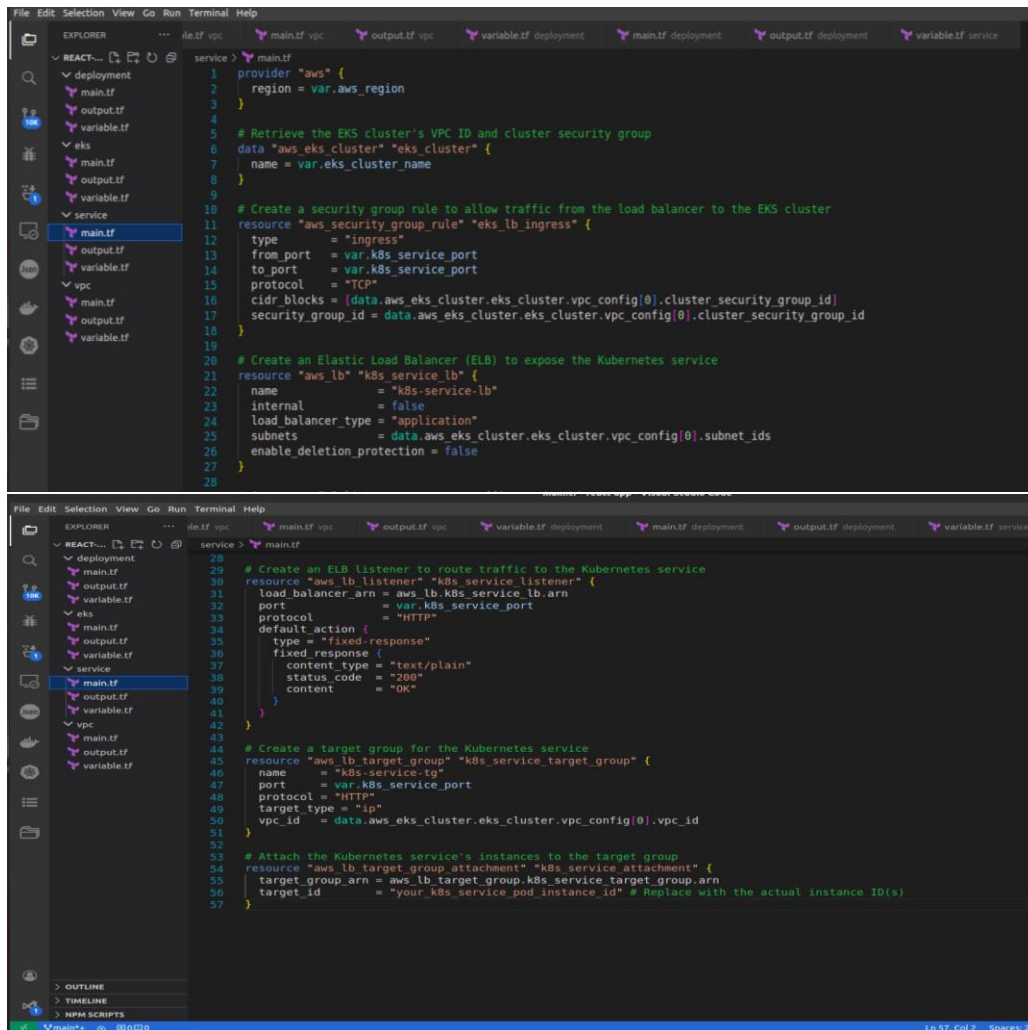
Step 4: Create Kubernetes Manifests with Terraform

Create Terraform configuration files (e.g., deployment.tf and service.tf) to define your Kubernetes resources.

deployment.tf:



service.tf:



Step 5: Apply the Terraform Configuration

Run the following commands to initialize Terraform and apply the configuration:

terraform init
terraform apply

Terraform will create the necessary Kubernetes resources based on your configuration.

Step 6: Access Your React Application

After applying the Terraform configuration, it may take some time for AWS to provision a load balancer for your service. You can check the status using `kubectl get svc my-react-app-service`. Once it has an external IP, you can access your React application using that IP.

Step 7: Scaling and Maintenance

You can scale your application by adjusting the replica count in the Deployment configuration ([deployment.tf](#)) and reapplying the Terraform configuration. For updates or maintenance, you can redeploy a new version of your Docker image and update the Deployment.

Conclusion

This documentation has outlined the process of deploying a React application on EKS using Terraform. Customize the Terraform configuration to match your specific requirements and best practices.

