Dakoju Rochan
rochan00003@gmail.com

Project

Deploying an EKS Cluster with Terraform and Deploying a BlogApp using custom Helm and Nginx Ingress Controller

BY

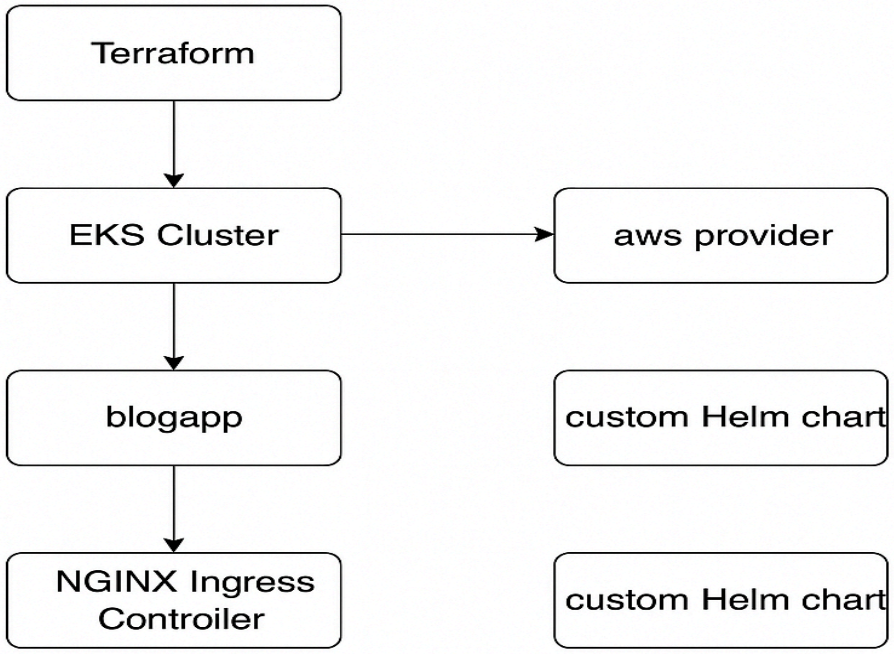
Rochan Dakoju

Dakoju Rochan
rochan00003@gmail.com

Prerequisite:

- Install AWS CLI LATEST VERSION
- INSTALL TERRAFORM
- INSTALL HELM
- INSTALL KUBECTL
- CREATE A EC2 INSTANCE
- CREATE A USER AND GENERATE A ACCESS KEY ADD THE PERMISSIONS TO THAT USER

Architectural Diagram:



This diagram illustrates a typical deployment process for a web application ("blogapp") on an Amazon Elastic Kubernetes Service (EKS) cluster using Infrastructure as Code (IaC) and a Kubernetes package manager. Let's break down each component and the flow:

1. Terraform:

- This is an Infrastructure as Code (IaC) tool. It allows you to define and provision infrastructure (like your EKS cluster) using a declarative configuration language.
- In this context, Terraform is likely used to create and manage the EKS cluster itself, including the control plane nodes, worker nodes, networking configurations (VPC, subnets, security groups), and IAM roles necessary for the cluster to function within AWS.
- The arrow pointing from "Terraform" to "EKS Cluster" signifies that Terraform is the tool used to provision and manage the lifecycle of the EKS cluster.

Dakoju Rochan
rochan00003@gmail.com

2. EKS Cluster:

- This is a managed Kubernetes service provided by Amazon Web Services (AWS). It simplifies the process of running Kubernetes on AWS without needing to install, operate, and maintain your own Kubernetes control plane.
- The EKS cluster provides the underlying platform where your applications and services will run as containerized workloads.

3. aws provider:

- This represents the AWS provider plugin for Terraform. To manage AWS resources like an EKS cluster, Terraform needs to interact with the AWS API. The AWS provider enables this interaction by providing the necessary resource definitions and functionalities.
- The arrow pointing from "EKS Cluster" to "aws provider" indicates that the EKS cluster relies on the AWS provider for its creation and management within the AWS environment.

4. blog app:

- This represents the actual web application that you want to deploy and run on your EKS cluster. It's likely packaged as a set of Docker containers.
- The arrow pointing from "EKS Cluster" to "blogapp" signifies that the "blogapp" is deployed and runs as pods (the smallest deployable units in Kubernetes) within the EKS cluster.

5. custom Helm chart (for blogapp):

- Helm is a package manager for Kubernetes. Helm charts define, install, and upgrade even the most complex Kubernetes applications.
- The "custom Helm chart" for "blogapp" contains all the necessary Kubernetes manifests (YAML files) to deploy and manage the "blogapp," such as Deployments, Services, ConfigMaps, Secrets, etc. It parameterizes the application deployment, making it easier to configure and upgrade.
-
- Although not explicitly connected by an arrow in this simplified diagram, the "blogapp" deployment within the EKS cluster would typically be orchestrated using this custom Helm chart.

6. NGINX Ingress Controller:

- An Ingress Controller is a type of Kubernetes controller that manages external access to the services within a Kubernetes cluster. NGINX Ingress Controller uses NGINX as a reverse proxy and load balancer.
- It listens for incoming HTTP and HTTPS requests and routes them to the appropriate services within the cluster based on rules defined in Ingress resources.
- The arrow pointing from "blogapp" to "NGINX Ingress Controller" suggests that the "blogapp" relies on the NGINX Ingress Controller to handle external traffic and make the application accessible from the internet.

7. custom Helm chart (for NGINX Ingress Controller):

- Similar to the "blog app," the NGINX Ingress Controller is also likely deployed and managed using a Helm chart. This simplifies the installation, configuration, and upgrading of the Ingress Controller within the EKS cluster.

In summary, the diagram illustrates the following workflow:

1. **Provisioning the Infrastructure:** Terraform, using the AWS provider, is used to create and manage the underlying EKS cluster on AWS.
2.
3. **Deploying the Application:** The "blogapp," packaged as container(s), is deployed onto the EKS cluster using a custom Helm chart. This chart defines how the application should run within Kubernetes.
4. **Exposing the Application:** The NGINX Ingress Controller, also deployed using a custom Helm chart, is configured to handle external traffic and route requests to the "blogapp" service running within the EKS cluster.

This setup provides a scalable, resilient, and well-managed environment for running the "blogapp" on AWS using Kubernetes best practices and infrastructure automation.

Dakoju Rochan
rochan00003@gmail.com

let's walk through the installation of the latest versions of AWS CLI, Terraform, Helm, and Kubectl. Since the current date is May 1, 2025, I'll provide instructions that should be relevant for installing the most up-to-date versions available at that time. Keep in mind that software versions evolve, so always refer to the official documentation for the absolute latest instructions if you encounter any issues.

Important Considerations Before You Begin:

- **Operating System:** The installation steps can vary significantly depending on your operating system (Windows, macOS, Linux). I will provide general instructions and highlight key differences. Please specify your operating system if you need more tailored guidance.
- **Permissions:** You might need administrator or sudo privileges on your system to install these tools.
- **Dependencies:** Some tools might have dependencies (e.g., Python for AWS CLI). I'll mention these where applicable.
- **Verification:** After each installation, I'll show you how to verify that the tool is installed correctly and check its version.

1. Install AWS Command Line Interface (CLI)

The AWS CLI is a command-line tool that enables you to interact with AWS services using commands in your shell.

General Steps (Specifics vary by OS):

- **Download the Installer:** Obtain the appropriate installer for your operating system from the official AWS documentation.
- **Run the Installer:** Execute the downloaded installer.
- **Add to Path (if necessary):** On some systems, you might need to manually add the AWS CLI executable to your system's PATH environment variable to access it from any terminal.

Specific Instructions (as of May 1, 2025 - check official AWS docs for the absolute latest):

- **Windows:**
    - Download the AWS CLI MSI installer from the official AWS documentation.
    - Run the downloaded MSI file and follow the on-screen instructions. The installer typically handles adding the CLI to your PATH.
    - Open a new Command Prompt or PowerShell window to ensure the PATH changes are applied.
- **macOS:**
    - **Using the Bundled Installer:** Download the macOS PKG installer from the official AWS documentation and run it.

Using pip (Python Package Installer): If you have Python and pip installed (which is common on macOS), you can use:
Bash
```
python3 -m pip install --upgrade --user awscli
```
After installation via pip, you might need to add the ~/.local/bin directory to your PATH. You can usually do this by adding the following line to your shell configuration file (e.g., ~/.bashrc, ~/.zshrc) and then sourcing the file:
Bash
```
export PATH="$HOME/.local/bin:$PATH"
source ~/.bashrc  # or source ~/.zshrc
```

    -
- **Linux:**

Using the Bundled Installer: AWS provides a bundled installer script. Download it using curl:
Bash
```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
```
Then, unzip and run the install script:
Bash

unzip awscliv2.zip
sudo ./aws/install

Using pip (if Python is available):
Bash
python3 -m pip install --upgrade awscli

- Depending on your Linux distribution, you might need to use pip3 instead of pip. You might also need to add the installation directory (often /usr/local/bin) to your PATH.

Verify AWS CLI Installation:

Open your terminal or command prompt and run:

Bash
aws --version

This command should output the installed AWS CLI version. If you see an error message like "command not found," it means the AWS CLI is not correctly installed or not in your system's PATH.

2. Install Terraform

Terraform is an Infrastructure as Code (IaC) tool that allows you to define and provision infrastructure across various cloud providers.

General Steps:

- **Download the Binary:** Terraform is distributed as a single binary executable. Download the appropriate binary for your operating system and architecture from the official HashiCorp website (the developers of Terraform).
- **Add to Path:** After downloading, it's highly recommended to add the Terraform executable to your system's PATH so you can run the terraform command from any directory.

Specific Instructions (as of May 1, 2025 - check official HashiCorp docs for the absolute latest):

- **Windows:**
    1. Go to the Terraform downloads page on the HashiCorp website.
    2. Download the ZIP file for your architecture (usually windows_amd64).
    3. Extract the terraform.exe file from the ZIP archive to a directory of your choice (e.g., C:\Terraform).
    4. Add this directory to your system's PATH environment variable. You can do this by searching for "environment variables" in the Start Menu, clicking "Edit the system environment variables," then clicking the "Environment Variables..." button. In the "System variables" section, find the "Path" variable, select it, and click "Edit...". Add the directory containing terraform.exe to the list.
    5.
    6. Open a new Command Prompt or PowerShell window.
- **macOS:**
    1. Go to the Terraform downloads page on the HashiCorp website.
    2. Download the ZIP file for macOS (usually darwin_amd64).

Extract the terraform executable from the ZIP archive to a directory such as /usr/local/bin (you might need sudo privileges for this).
Bash
unzip terraform_*.zip
sudo mv terraform /usr/local/bin/

    3.
    4. Ensure /usr/local/bin is in your system's PATH (it usually is by default).
- **Linux:**

1. Go to the Terraform downloads page on the HashiCorp website.
2. Download the ZIP file for your Linux distribution and architecture (e.g., linux_amd64).

Extract the terraform executable to a directory such as /usr/local/bin (you might need sudo privileges).
Bash
unzip terraform_*.zip
sudo mv terraform /usr/local/bin/

3.
4. Ensure /usr/local/bin is in your system's PATH.

Verify Terraform Installation:

Open your terminal or command prompt and run:

Bash
terraform --version

This should output the installed Terraform version.

3. Install Helm

Helm is a package manager for Kubernetes, allowing you to easily deploy and manage applications on your Kubernetes clusters.

General Steps:

- **Download the Binary:** Helm is distributed as a binary executable. Download the appropriate version for your operating system and architecture from the official Helm releases page on GitHub or the Helm website.
- **Add to Path:** Add the Helm executable (helm) to your system's PATH.

Specific Instructions (as of May 1, 2025 - check official Helm docs for the absolute latest):

- **Windows:**
    - Go to the Helm releases page on GitHub or the official Helm website.
    - Download the ZIP file for Windows (usually windows-amd64).
    - Extract the helm.exe file to a directory of your choice (e.g., C:\Helm).
    - Add this directory to your system's PATH environment variable (as described in the Terraform installation for Windows).
    - Open a new Command Prompt or PowerShell window.
- **macOS:**

Using Homebrew (recommended): If you have Homebrew installed, you can install Helm with:
Bash
brew install helm

    -

Manual Download: Download the macOS TAR.GZ file from the Helm releases page.
Bash
tar -zxvf helm-vX.Y.Z-darwin-amd64.tar.gz # Replace X.Y.Z with the actual version
cd helm-vX.Y.Z-darwin-amd64
sudo mv darwin-amd64/helm /usr/local/bin/

    - Make sure /usr/local/bin is in your PATH.
- **Linux:**
    - **Using Package Managers (e.g., apt, yum):** The Helm documentation often provides instructions for

installing via package managers for various distributions. Check the official Helm installation guide.

Manual Download: Download the Linux TAR.GZ file from the Helm releases page.
Bash
```
tar -zxvf helm-vX.Y.Z-linux-amd64.tar.gz  # Replace X.Y.Z with the actual version
cd helm-vX.Y.Z-linux-amd64
sudo mv linux-amd64/helm /usr/local/bin/
```

- Make sure /usr/local/bin is in your PATH.

Verify Helm Installation:

Open your terminal or command prompt and run:

Bash
```
helm version
```

This should output the client and server (if you have a Tiller component running in your cluster - note that Helm v3 and later do not use Tiller by default) versions.

4. Install Kubectl

Kubectl is the command-line tool for interacting with Kubernetes clusters. It allows you to run commands against Kubernetes clusters.

General Steps:

- **Download the Binary:** Download the appropriate kubectl binary for your operating system and architecture from the official Kubernetes documentation or a trusted source.
- **Add to Path:** Add the kubectl executable to your system's PATH.

Specific Instructions (as of May 1, 2025 - check official Kubernetes docs for the absolute latest):

- **Windows:**
  - Go to the Kubernetes documentation and find the instructions for installing kubectl on Windows. You'll typically download a kubectl.exe file.
  - Download the binary to a directory of your choice (e.g., C:\Kubectl).
  - Add this directory to your system's PATH environment variable (as described in the Terraform installation for Windows).
  - Open a new Command Prompt or PowerShell window.
  - You might also want to install choco (Chocolatey package manager for Windows) which can simplify the installation: choco install kubernetes-cli.
- **macOS:**

Using Homebrew (recommended):
Bash
```
brew install kubectl
```

-

Manual Download: Download the kubectl binary from the Kubernetes documentation. Make it executable and move it to a directory in your PATH:
Bash
```
curl -LO "https://dl.k8s.io/release/vX.YY/bin/darwin/amd64/kubectl"  # Replace vX.YY with the latest stable version
chmod +x kubectl
sudo mv kubectl /usr/local/bin/
```

- **Linux:**

- **Using Package Managers (e.g., apt, yum):** The Kubernetes documentation provides instructions for installing kubectl using package managers for various distributions.

Manual Download: Download the kubectl binary:

curl -LO "https://dl.k8s.io/release/vX.YY/bin/linux/amd64/kubectl" # Replace vX.YY with the latest stable version
chmod +x kubectl
sudo mv kubectl /usr/local/bin/

Verify Kubectl Installation:

Open your terminal or command prompt and run:

kubectl version --client

This command should output the client version of kubectl. If you have a Kubernetes cluster configured, you can also run the kubectl version to see both the client and server versions.

1. Create a Terraform main.tf File for Creating EKS Cluster

```
provider "aws" { region
  = "ap-south-1"
}

# Fetch default VPC
data "aws_vpc" "default" {
  default = true
}

# Fetch default subnets
data "aws_subnets" "default" {
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}

# Security Group for EKS
resource "aws_security_group" "eks_sg" {
  vpc_id = data.aws_vpc.default.id

  ingress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
```

Dakoju Rochan
rochan00003@gmail.com

```
    Name = "eks-security-group"
  }
}

# IAM Role for EKS Cluster
resource "aws_iam_role" "eks_role" {
  name = "eks-cluster-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17" Statement
    = [{
      Action = "sts:AssumeRole"
      Effect = "Allow"
      Principal = {
        Service = "eks.amazonaws.com"
      }
    }]
  })
}

# Attach necessary IAM policies to EKS Role
resource "aws_iam_role_policy_attachment" "eks_policy_attach" {
```

```
  role       = aws_iam_role.eks_role.name
  policy_arn = "arn:aws:iam::aws:policy/AmazonEKSClusterPolicy"
}

# Create EKS Cluster
resource "aws_eks_cluster" "eks_cluster" {
  name     = "my-eks-cluster"
  role_arn = aws_iam_role.eks_role.arn

  vpc_config {
    subnet_ids         = data.aws_subnets.default.ids
    security_group_ids = [aws_security_group.eks_sg.id]
  }
}

# Output EKS Cluster Details
output "cluster_name" {
  value = aws_eks_cluster.eks_cluster.name
}
```

Run the following Terraform commands:

```
terraform init
terraform validate
terraform plan
terraform apply -auto-approve
```

Configure Kubernetes Access:

```
aws eks update-kubeconfig --region ap-south-1 --name my-eks-cluster
```

Verify Cluster Details:

```
kubectl get nodes -A
kubectl get all -A
```

2. Create a Custom Helm Chart for Deploying BlogApp

Create Helm Chart:

```
helm create blogapp-chart
```

Directory Structure:

```
blogapp-chart/
  ├── templates/
  │     ├── deployment.yaml
  │     ├── service.yaml
  ├── values.yaml
  ├── Chart.yaml
```

Define Custom Deployment (`deployment.yaml`):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}
  namespace: {{ .Values.namespace }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ .Values.appName }}
  template:
    metadata:
      labels:
        app: {{ .Values.appName }}
    spec:
      containers:
      - name: {{ .Values.appName }}
        image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
        ports:
        - containerPort: {{ .Values.service.port }}
```

Define Service (`service.yaml`):

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}-svc
  namespace: {{ .Values.namespace }}
spec:
  selector:
    app: {{ .Values.appName }}
  ports:
    - protocol: TCP
      port: {{ .Values.service.port }} targetPort:
      {{ .Values.service.targetPort }}
```

Define Values (`values.yaml`):

```
appName: blogapp
namespace: webapps
replicaCount: 1

image:
  repository: <your-repository>
  tag: "latest"

service:
  port: 8080
  targetPort: 8080
```

Install the Helm Chart:

```
helm install blogapp ./blogapp-chart -n webapps
kubectl get all -n webapps
```

3. Expose BlogApp Externally using Nginx Ingress Controller

Install Nginx Ingress Controller:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
helm install nginx-ingress ingress-nginx/ingress-nginx --namespace ingress-nginx --create-namespace
```

Check Ingress Controller Status:

```
kubectl get all -n ingress-nginx
```

Create an Ingress Resource (`ingress.yaml`):

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: blogapp-ingress
  namespace: webapps
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - host: test.example.com
    http:
      paths:
      - path: /blog
        pathType: Prefix
        backend:
          service:
            name: blogapp-svc
            port:
              number: 8080
```

Apply Ingress Configuration:

```
kubectl apply -f ingress.yaml
```

Verify Ingress:

```
kubectl get ingress -n webapps
```

Get External IP Address:

```
nslookup <aws-eks-endpoint>
```

Test Locally:

Edit `/etc/hosts` `C:\Windows\System32\drivers\etc\hosts` Linux/macOS) or (Windows):

```
<INGRESS-EXTERNAL-IP> test.example.com
```

Now your BlogApp is accessible externally!