**Prometheus-Terraform**

This document explains, step by step, what each Terraform block does, why it’s needed, where it fetches things from, how to verify, and how to fix common issues. It follows the exact structure you requested.

1) Connecting to AWS & the EKS Cluster

**Code:**

provider "aws" {  
 region = var.aws\_region  
}  
  
data "aws\_eks\_cluster" "eks" { name = var.eks\_cluster\_name }  
data "aws\_eks\_cluster\_auth" "eks" { name = var.eks\_cluster\_name }  
  
provider "kubernetes" {  
 host = data.aws\_eks\_cluster.eks.endpoint  
 cluster\_ca\_certificate = base64decode(data.aws\_eks\_cluster.eks.certificate\_authority[0].data)  
 token = data.aws\_eks\_cluster\_auth.eks.token  
}  
  
provider "helm" {  
 kubernetes {  
 host = data.aws\_eks\_cluster.eks.endpoint  
 cluster\_ca\_certificate = base64decode(data.aws\_eks\_cluster.eks.certificate\_authority[0].data)  
 token = data.aws\_eks\_cluster\_auth.eks.token  
 }  
}

**Summary:**

* Connects Terraform to AWS in your region.
* It Reads EKS cluster endpoint and certificate and fetches a short-lived token.
* Configures Kubernetes and Helm providers to manage resources directly inside EKS cluster.

**What it does**

* Use the AWS provider to talk to AWS APIs in selected region.
* It Reads EKS cluster’s API endpoint and CA and requests a short‑lived auth token.
* Configures the Kubernetes and Helm providers so Terraform can create Kubernetes objects and install Helm charts.

**Why it’s needed**

* Without endpoint/CA/token, Terraform cannot safely create resources inside the cluster.

**Inputs / Outputs**

* Inputs: var.aws\_region, var.eks\_cluster\_name.
* Output: in‑memory connection details for the Kubernetes and Helm providers (no resources created).

**Verify**

* aws eks describe-cluster --name <cluster> should show the same endpoint Terraform uses.
* kubectl get ns should work from your workstation (same cluster/context).

**Common issues & fixes**

* Expired/invalid token: re‑run aws eks update-kubeconfig or ensure the caller has EKS DescribeCluster permission.
* Wrong cluster name/region: double‑check variables.

2) Setting Up the OIDC Provider (IRSA foundation)

**Code:**

resource "aws\_iam\_openid\_connect\_provider" "eks\_oidc" {  
 client\_id\_list = ["sts.amazonaws.com"]  
 thumbprint\_list = ["9e99a48a9960b14926bb7f3b02e22da0afd10df6"]  
 url = data.aws\_eks\_cluster.eks.identity[0].oidc[0].issuer  
}

**Summary:**

* Creates an IAM OIDC provider linked to our EKS cluster.
* Enables IRSA so pods can assume IAM roles without static keys.

**What it does**

* Registers your cluster’s OIDC issuer URL with IAM as a trusted identity provider.
* Whitelists the OIDC audience/client sts.amazonaws.com and pins the TLS thumbprint.

**Why it’s needed**

* Enables IAM Roles for Service Accounts (IRSA) for least‑privilege, short‑lived credentials at pod level.

**Inputs / Outputs**

* Inputs: EKS OIDC issuer, thumbprint, client\_id\_list=['sts.amazonaws.com'].
* Output: An IAM OIDC provider ARN used by the role trust policy (next step).

**Verify**

* IAM → Identity providers shows an OpenID Connect provider for your issuer.
* aws eks describe-cluster --query 'cluster.identity.oidc.issuer' matches the provider URL.

**Common issues & fixes**

* Issuer mismatch/typo: Ensure exact URL match (no trailing slash).
* Wrong thumbprint: Update when the issuer’s CA chain changes.
* Duplicate provider: Reuse existing; don’t create a second for the same issuer.

3) ALB Controller IAM Policy & Role (with trust)

**Code (policy, role, attachment):**

resource "aws\_iam\_policy" "alb\_controller" {  
 name = "AWSLoadBalancerControllerIAMPolicy"  
 policy = file("${path.module}/alb-iam-policy.json")  
}  
  
resource "aws\_iam\_role" "alb\_controller" {  
 name = "alb-controller-role"  
 assume\_role\_policy = jsonencode({  
 Version = "2012-10-17",  
 Statement = [{  
 Effect = "Allow",  
 Principal = { Federated = aws\_iam\_openid\_connect\_provider.eks\_oidc.arn },  
 Action = "sts:AssumeRoleWithWebIdentity",  
 Condition = {  
 StringEquals = {  
 "<OIDC\_ISSUER>:sub" = "system:serviceaccount:kube-system:aws-load-balancer-controller"  
 }  
 }  
 }]  
 })  
}  
  
resource "aws\_iam\_role\_policy\_attachment" "alb\_attach" {  
 role = aws\_iam\_role.alb\_controller.name  
 policy\_arn = aws\_iam\_policy.alb\_controller.arn  
}

**Summary:**

* Loads a least‑privilege IAM policy from alb-iam-policy.json.
* Creates an IAM role that only the Kubernetes service account aws-load-balancer-controller can assume via IRSA.
* Attaches the policy so the controller can manage ALBs, target groups, listeners, and tags.

**What it does**

* Defines necessary AWS permissions for the controller (EC2 describes, ELBv2 actions, tagging).
* Locks the trust policy to a single service account subject for strong isolation.

**Why it’s needed**

* The controller must create/manage AWS load balancer resources on your behalf.

**Inputs / Outputs**

* Inputs: local alb-iam-policy.json; OIDC provider ARN from Step 2.
* Outputs: IAM role ARN used by the Kubernetes service account in Step 4.

**Verify**

* IAM → Roles → alb-controller-role has the correct trust (aud/sub) and attached policy.
* No overly broad actions in policy (least‑privilege review).

**Common issues & fixes**

* Service account name/namespace changed: update the trust policy subject.
* Missing actions: add only the minimal required permissions; avoid wildcards where possible.

4) Kubernetes Service Account for the ALB Controller (IRSA binding)

**Code:**

resource "kubernetes\_service\_account" "alb\_sa" {  
 metadata {  
 name = "aws-load-balancer-controller"  
 namespace = "kube-system"  
 annotations = {  
 "eks.amazonaws.com/role-arn" = aws\_iam\_role.alb\_controller.arn  
 }  
 }  
}

**Summary:**

* Creates the service account used by the ALB Controller deployment in kube-system.
* Annotates it with the IAM Role ARN, enabling IRSA.

**What it does**

* Binds Kubernetes identity (service account) to AWS IAM role securely via web identity tokens.

**Why it’s needed**

* Allows the controller pod to obtain temporary AWS credentials without static keys.

**Inputs / Outputs**

* Input: IAM role ARN from Step 3.
* Output: ServiceAccount object in Kubernetes.

**Verify**

* kubectl -n kube-system get sa aws-load-balancer-controller -o yaml shows the eks.amazonaws.com/role-arn annotation.

**Common issues & fixes**

* Annotation typo/wrong ARN: controller pods receive AccessDenied; fix annotation and restart.

5) Install the AWS Load Balancer Controller (Helm)

**Code:**

resource "helm\_release" "alb\_controller" {  
 name = "alb-controller-ci"  
 namespace = "kube-system"  
 repository = "https://aws.github.io/eks-charts"  
 chart = "aws-load-balancer-controller"  
 version = "1.7.1"  
  
 set { name = "clusterName"; value = var.eks\_cluster\_name }  
 set { name = "region"; value = var.aws\_region }  
 set { name = "vpcId"; value = var.vpc\_id }  
 set { name = "autoDiscoverSubnets"; value = "true" }  
 set { name = "serviceAccount.create"; value = "false" }  
 set { name = "serviceAccount.name"; value = "aws-load-balancer-controller" }  
}

**Summary:**

* Downloads the controller chart from AWS EKS charts repo and installs it into kube-system.
* Reuses the pre-created service account; enables auto-discovery of subnets via tags.

**What it does**

* Deploys the controller that watches Ingress/Service resources and provisions ALBs automatically.

**Why it’s needed**

* This controller is the bridge between Kubernetes Ingress and AWS ALB resources.

**Inputs / Outputs**

* Inputs: clusterName, region, vpcId, serviceAccount settings.
* Outputs: Deployment/Pods in kube-system; CRDs/IngressClass as provided by the chart.

**Verify**

* kubectl -n kube-system get deploy,pods | grep aws-load-balancer-controller shows Running/Ready.

**Common issues & fixes**

* IRSA not working: check SA annotation and role trust.
* No suitable subnets discovered: ensure Step 7 tags exist.

6) Finding Private Subnets (per AZ)

**Code (preferred implementation):**

data "aws\_subnets" "all\_in\_vpc" {  
 filter { name = "vpc-id" values = [var.vpc\_id] }  
}  
  
data "aws\_subnet" "details" {  
 for\_each = toset(data.aws\_subnets.all\_in\_vpc.ids)  
 id = each.value  
}  
  
locals {  
 private\_subnet\_ids = [  
 for az in distinct([  
 for s in data.aws\_subnet.details : s.availability\_zone if s.map\_public\_ip\_on\_launch == false  
 ]) :  
 sort([  
 for id, s in data.aws\_subnet.details :  
 id if s.availability\_zone == az && s.map\_public\_ip\_on\_launch == false  
 ])[0]  
 ]  
}

**Summary:**

* Reads all subnets in the VPC, filters to private ones (no public IP on launch).
* Selects one private subnet per Availability Zone for ALB high availability.

**What it does**

* Builds a deterministic list of private subnets distributed across AZs.

**Why it’s needed**

* Internal ALBs must run in private subnets; multi‑AZ improves resilience.

**Inputs / Outputs**

* Input: var.vpc\_id.
* Output: local.private\_subnet\_ids (one per AZ).

**Verify**

* VPC console → Subnets → confirm 'Auto-assign public IPv4 address' is disabled for chosen subnets.

7) Tagging Subnets for Load Balancers

**Code:**

resource "aws\_ec2\_tag" "internal\_elb\_tag" {  
 for\_each = toset(local.private\_subnet\_ids)  
 resource\_id = each.value  
 key = "kubernetes.io/role/internal-elb"  
 value = "1"  
}  
  
resource "aws\_ec2\_tag" "cluster\_tag" {  
 for\_each = toset(local.private\_subnet\_ids)  
 resource\_id = each.value  
 key = "kubernetes.io/cluster/${var.eks\_cluster\_name}"  
 value = "owned"  
}

**Summary:**

* Applies required subnet tags so the controller can place \*\*internal\*\* ALBs and associate them with this cluster.

**What it does**

* Marks subnets as eligible for internal ALBs and owned by the cluster.

**Why it’s needed**

* The controller discovers allowed subnets via these tags (autoDiscoverSubnets=true).

**Verify**

* VPC → Subnets → Tags show kubernetes.io/role/internal-elb=1 and kubernetes.io/cluster/<name>=owned on each selected subnet.

8) Creating the Monitoring Namespace

**Code:**

resource "kubernetes\_namespace" "monitoring" {  
 metadata { name = "demo-monitoring" }  
}

**Summary:**

* Creates a dedicated namespace for Prometheus, Grafana, and Alertmanager.

**Verify**

* kubectl get ns demo-monitoring

9) Installing Prometheus & Grafana (kube‑prometheus‑stack)

**Code:**

resource "helm\_release" "prometheus\_stack" {  
 name = "kube-prometheus-stack"  
 repository = "https://prometheus-community.github.io/helm-charts"  
 chart = "kube-prometheus-stack"  
 version = "56.6.2"  
 namespace = kubernetes\_namespace.monitoring.metadata[0].name  
 values = [file("${path.module}/values.yaml")]  
}

**Summary:**

* Downloads the chart from the Prometheus Community repo and deploys Prometheus, Grafana, and Alertmanager into demo-monitoring.
* Uses your local values.yaml for configuration (ClusterIP services, Grafana port 80, etc.).

**What it does**

* Installs a full monitoring stack ready to be exposed via the ALB Ingress.

**Verify**

* kubectl -n demo-monitoring get pods,svc shows the components running.

10) Creating the Ingress (Internal ALB)

**Code:**

resource "kubernetes\_ingress\_v1" "monitoring\_ingress" {  
 metadata {  
 name = "monitoring-ingress"  
 namespace = "demo-monitoring"  
 annotations = {  
 "alb.ingress.kubernetes.io/scheme" = "internal"  
 "alb.ingress.kubernetes.io/target-type" = "ip"  
 "alb.ingress.kubernetes.io/listen-ports" = "[{\"HTTP\":80}]"  
 "alb.ingress.kubernetes.io/load-balancer-attributes" = "idle\_timeout.timeout\_seconds=60"  
 "alb.ingress.kubernetes.io/group.name" = "monitoring"  
 "alb.ingress.kubernetes.io/group.order" = "10"  
 "alb.ingress.kubernetes.io/subnets" = join(",", local.private\_subnet\_ids)  
 }  
 }  
 spec {  
 ingress\_class\_name = "alb"  
 rule {  
 http {  
 path {  
 path = "/grafana"  
 path\_type = "Prefix"  
 backend { service { name = "kube-prometheus-stack-grafana"; port { number = 80 } } }  
 }  
 path {  
 path = "/prometheus"  
 path\_type = "Prefix"  
 backend { service { name = "kube-prometheus-stack-prometheus"; port { number = 9090 } } }  
 }  
 }  
 }  
 }  
}

**Summary:**

* Declares an Ingress that instructs the controller to build an \*\*internal\*\* ALB across your private subnets.
* Routes /grafana and /prometheus paths to the correct services.

**What it does**

* Creates an AWS Application Load Balancer, listeners, rules, and target groups via the controller.

**Verify**

* kubectl -n demo-monitoring get ingress monitoring-ingress -o wide shows an ALB hostname.
* EC2 → Load Balancers shows an internal ALB with two rules; target groups report healthy.

11) Wait for ALB Controller Readiness (race-condition guard)

**Code:**

resource "null\_resource" "wait\_for\_alb\_controller" {  
 provisioner "local-exec" {  
 command = <<EOT  
 kubectl wait --for=condition=Ready pod -l app.kubernetes.io/name=aws-load-balancer-controller -n kube-system --timeout=300s  
 EOT  
 }  
}

**Summary:**

* Ensures the controller pod is Ready before creating the Ingress to avoid race conditions.

12) Variables & Versions

**Code:**

variable "aws\_region" { default = "us-east-1" }  
variable "eks\_cluster\_name" { default = "test-kube" }  
variable "vpc\_id" { type = string }  
  
terraform {  
 required\_version = ">= 1.3.0"  
 required\_providers {  
 aws = { source = "hashicorp/aws", version = "~> 5.0" }  
 kubernetes = { source = "hashicorp/kubernetes", version = "~> 2.0" }  
 helm = { source = "hashicorp/helm", version = "~> 2.0" }  
 }  
}

**Summary:**

* Declares input variables and pins provider versions for reproducibility.

What Gets Downloaded & From Where

* Helm charts: ALB Controller from https://aws.github.io/eks-charts
* Helm charts: kube-prometheus-stack from https://prometheus-community.github.io/helm-charts
* Local files: alb-iam-policy.json and values.yaml from your Terraform module directory

Execution Flow (Apply Order)

1. Connect to AWS & EKS (providers + data sources).
2. Create OIDC Provider (IRSA foundation).
3. Create IAM Policy & Role for ALB Controller; attach policy.
4. Create Kubernetes ServiceAccount annotated with role (IRSA binding).
5. Install ALB Controller via Helm; wait until Ready.
6. Discover private subnets and tag them for internal ALB + cluster ownership.
7. Create demo-monitoring namespace.
8. Install kube-prometheus-stack via Helm.
9. Create Ingress → controller provisions internal ALB with /grafana and /prometheus paths.