Infrastructure-as-Code Driven Cloud-Native Three-Tier Application on AWS EKS

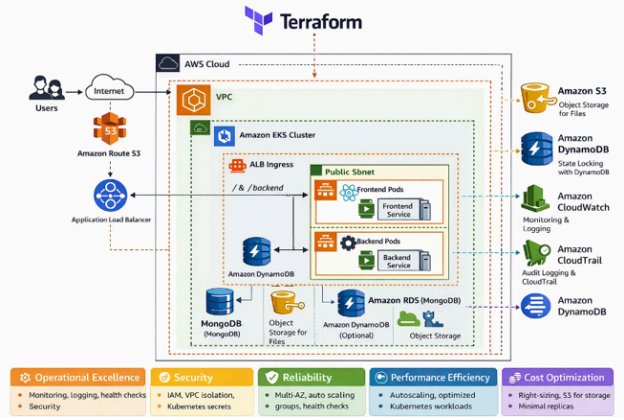


Figure 1: AWS Well-Architected three-tier cloud-native application deployed on Amazon EKS using Terraform, ALB Ingress Controller, S3 remote state, and DynamoDB locking.

📌 Overview

This diagram illustrates a **cloud-native three-tier application** deployed on **Amazon EKS**, fully provisioned using **Terraform** and aligned with the **AWS Well-Architected Framework**.

The solution demonstrates production-grade practices including Infrastructure as Code, Kubernetes ingress management, scalable application design, and secure state management.

🧱 Architecture Components

Infrastructure & Platform

- **Amazon VPC** – Isolated networking with public and private subnets
- **Amazon EKS** – Managed Kubernetes control plane
- **Managed Node Groups** – Auto-scaling worker nodes
- **Terraform** – Infrastructure provisioning and lifecycle management
- **S3 (Remote State)** – Centralized Terraform state storage
- **DynamoDB** – Terraform state locking to prevent race conditions

Application Layer

- **Frontend (React / Node.js)** – Served via Kubernetes Service
- **Backend (API)** – REST API exposed internally
- **MongoDB** – Persistent data storage inside the cluster

Traffic & Networking

- **AWS Load Balancer Controller**
- **Application Load Balancer (ALB)**
- **Path-based routing**
  - `/` → Frontend service
  - `/backend` → Backend service (path rewritten at ingress)

🧭 AWS Well-Architected Framework Alignment

🟦 Operational Excellence

- Terraform-driven infrastructure

- Declarative Kubernetes manifests
- Repeatable deployments and rollbacks

🟩 **Security**

- IAM roles for service accounts (IRSA)
- Private subnets for workloads
- No hard-coded secrets (Kubernetes Secrets)

🟨 **Reliability**

- Health checks (liveness & readiness probes)
- ALB target group health monitoring
- Self-healing Kubernetes pods

🟧 **Performance Efficiency**

- Horizontal scalability via Kubernetes
- ALB path-based routing
- Managed EKS control plane

🟥 **Cost Optimization**

- On-demand resources only
- No over-provisioned services
- Terraform destroy for clean teardown

🧠 **Challenges & Learnings**

- Fixed **ImagePullBackOff** issues
- Resolved **ALB target group health check failures**
- Implemented **Ingress path rewriting without changing application code**
- Debugged **Kubernetes rollout and deployment selector immutability**

Key Highlights

- End-to-end **Terraform + Kubernetes** deployment
- **Ingress path rewriting** without modifying containers
- Real-world troubleshooting (ImagePullBackOff, ALB health checks)
- Production-ready architecture

Executive Summary – AWS EKS Three-Tier Application

This project showcases a **cloud-native three-tier application** deployed on **Amazon EKS**, designed and implemented following **AWS Well-Architected Framework principles**.

The infrastructure is fully automated using **Terraform**, with state managed securely via **Amazon S3** and **DynamoDB locking**. Application traffic is handled through an **Application Load Balancer (ALB)** using **path-based routing and ingress rewrite rules**, enabling clean separation of frontend and backend services without modifying application code.

Key operational challenges such as **container image pull failures**, **ingress health check mismatches**, and **Kubernetes rollout issues** were identified and resolved using industry best practices. The final system is **scalable, resilient, secure, and cost-efficient**, reflecting real-world cloud engineering scenarios.

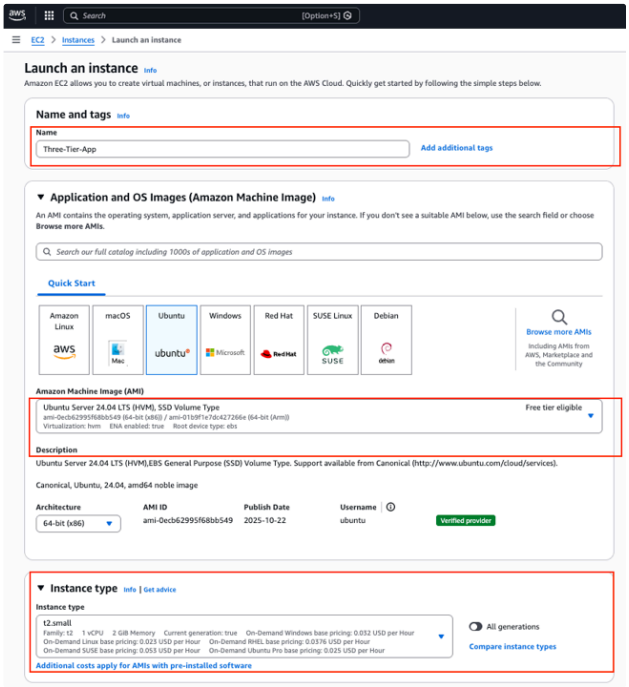
This project demonstrates hands-on expertise in:

- AWS networking and EKS
- Kubernetes operations and ingress design
- Terraform Infrastructure as Code
- Production troubleshooting and observability
- Well-Architected cloud design

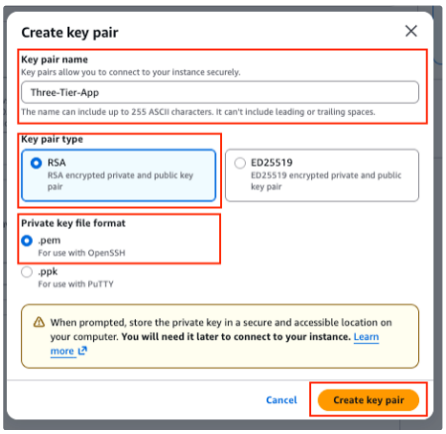
Step 1: Set up EC2 Instance

This step can be automated using Terraform or CloudFormation in AWS. However, I chose this as we are only setting up one EC2 instance and not multiple EC2 instances.

Note : As a future enhancement – we can update the code in Terraform to launch the EC2 instance automatically.

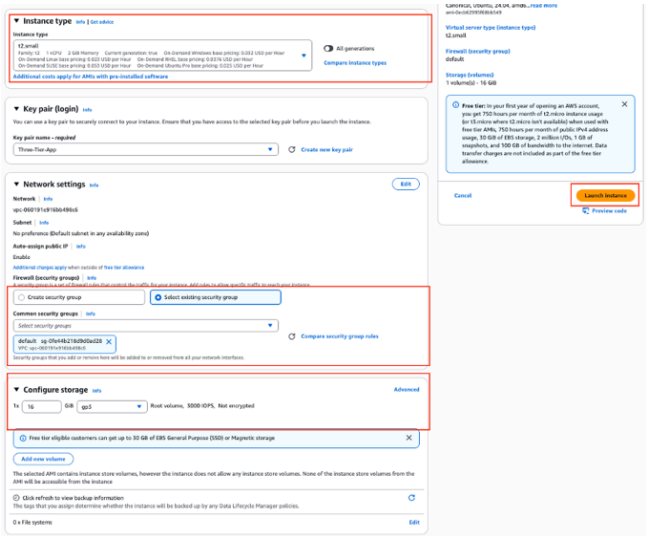


- The next step is to create the keypair as **Three-Tier-App**



Once we enter this – a **Three-Tier-App.pem** file will be automatically downloaded in your browser.

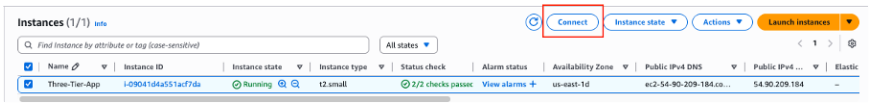
- Now let's select an existing security group and select from the drop down options
- Configure storage at 16Gib
- Click on **Launch Instance**



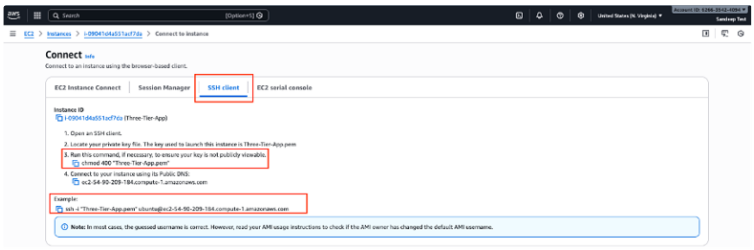
Step 2: Connect to the EC2 Instance remotely

Since we downloaded the PEM file we can now connect to the instance.

We need to retrieve the IP address that we need to connect so we will go to the EC2 Instance and click on **Connect**



So we will now have the instructions needed to connect to the EC2 instance via SSH.

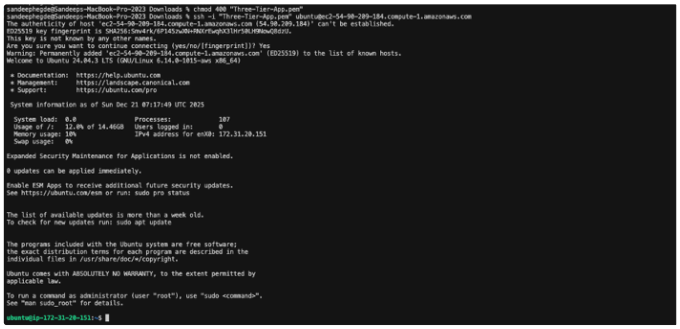


Let's keep these commands handy and proceed to VS code

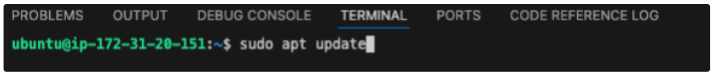
Tip: Ensure the *.pem file is in the same folder while connecting. For this demo- The file was downloaded to Downloads so I've navigated to Downloads folder

Advanced Tip: For security reasons – I've edited my Inbound Rules for this security group to only allow access from my IP rather than 0.0.0.0/0.

Since this project is less than a few hours – It's okay to keep it like this.



We will now install the necessary updates for this Ubuntu machine



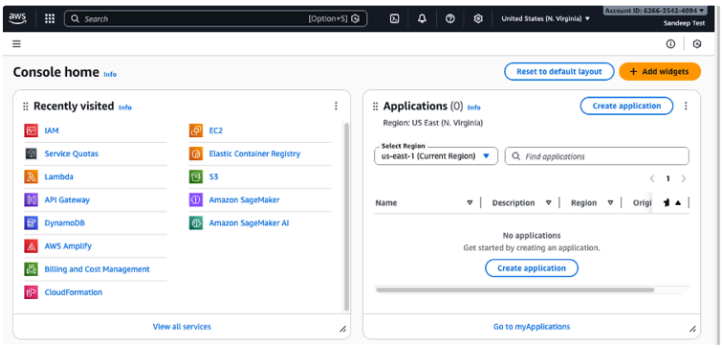
Step 3: Install Docker Container

We will now install Docker for this instance

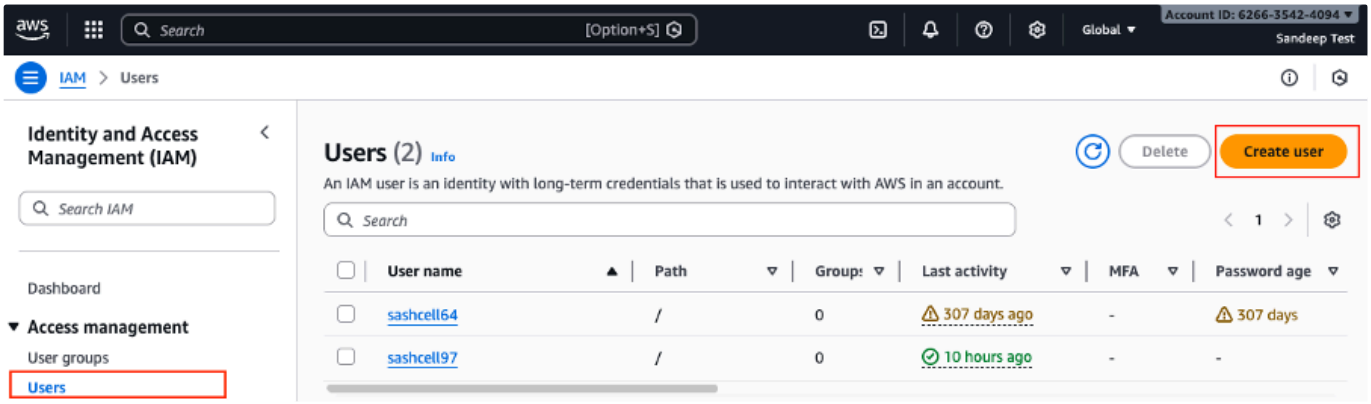
```
sudo apt install docker.io -y && sudo usermod -aG docker $USE && newgrp docker
```

Step 4: Set up AWS CLI and IAM User, Connect to the host machine

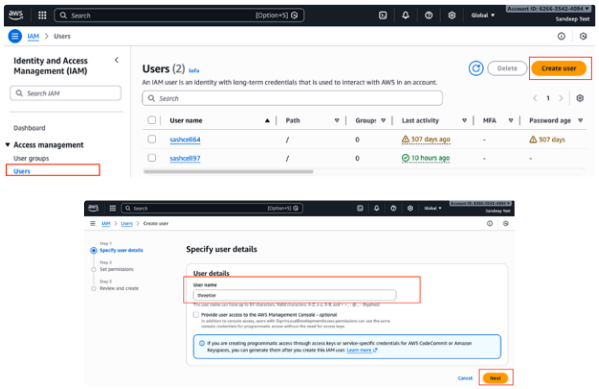
Navigate to AWS Console and click on IAM



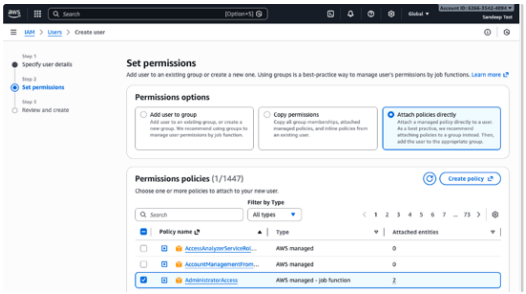
Now click on **Users** and click on **Create User**



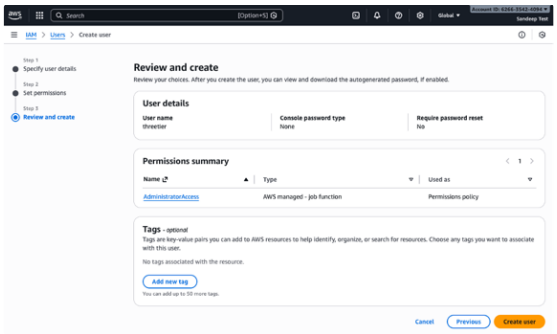
Next Step add a Name for the User and then click on Next



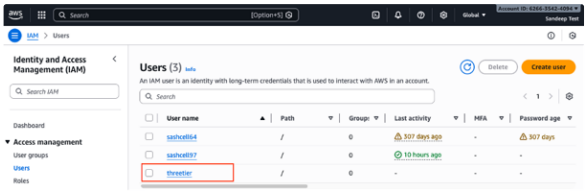
Now select the level of Permissions you need for access. We've selected **AdministratorAccess** however this depends in an enterprise or corporate setup. Then click on Next



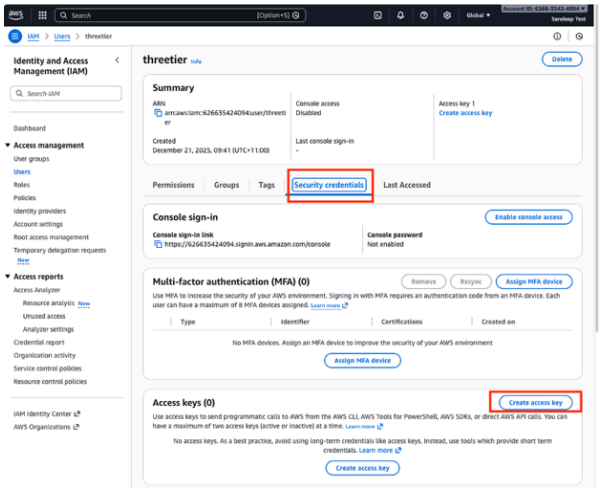
Review the options and then click on **Create User**



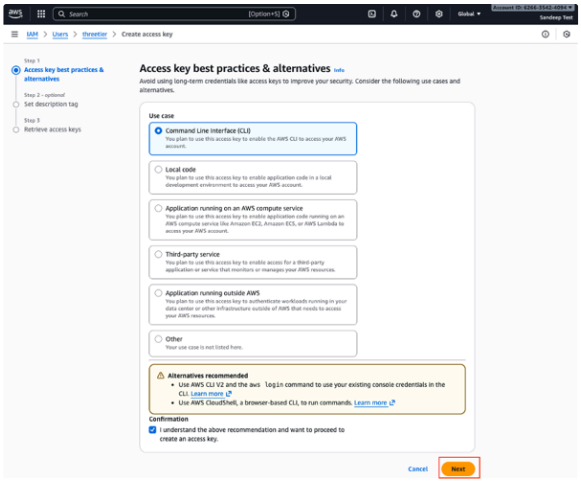
Once this is setup – we need to ensure that we can provide CLI access – Hence for this we now need to click on the newly created Administrator access



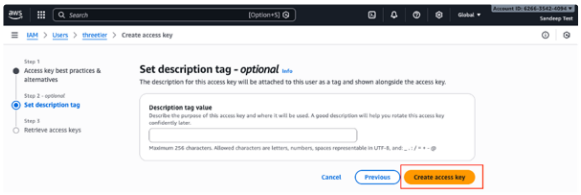
Select **Security Credentials** and then click on **Create access key**



Now select **Command Line Interface**, select the Confirmation box and then click on **Next**



Move on to the next step and click on **Create Access Key**



Now we will have the Access Keys. We need to make sure that we keep these keys secure and safe.

We will now need to download AWS CLI on our machine

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
sudo apt install -y unzip curl
unzip awscliv2.zip
sudo ./aws/install
```

Check the AWS version by going to

```
aws --version
```

We will now connect to the host machine securely

```
aws configure
```

It will prompt you to enter

- **Access Key ID**
- **Secret Access Key**
- **Default region name**: You can use `us-east-1` for this demo
- **Default output format**: Enter `json`

Once this is done we are now ready for the next step.

Step 5 : Install Terraform and set up remote Backend

Since the AWS CLI has been installed and configured we will now install Terraform on the backend.

```
sudo apt-get update && sudo apt-get install -y gnupg software-properties-common

wget -O- https://apt.releases.hashicorp.com/gpg | \
gpg --dearmor | \
sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg > /dev/null

echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \
https://apt.releases.hashicorp.com $(grep -oP '(?<=UBUNTU_CODENAME=).*' /etc/os-release || lsb_release -cs) main" | \
```

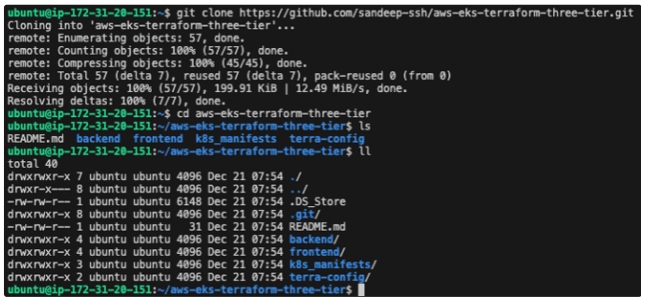
```
sudo tee /etc/apt/sources.list.d/hashicorp.list

sudo apt update

sudo apt-get install terraform
```

Let's validate the install now

```
terraform -v
```

Let's clone the repository and listing the contents we will see the below

```
git clone https://github.com/sandeep-ssh/aws-eks-terraform-three-tier.git
```



Terraform tracks the state of your infrastructure in a file called `terraform.tfstate`. By default, this state is stored locally, which is neither secure nor scalable. To follow Terraform best practices, we configure a **remote backend using Amazon S3**, ensuring safe, centralized, and collaborative state management.

We will now create a S3 bucket for the Terraform backend.

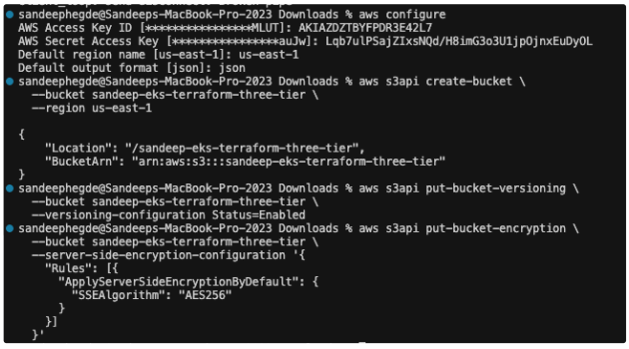
```
aws s3api create-bucket \
  --bucket sandeep-eks-terraform-three-tier \
  --region us-east-1
```

Enabling versioning so we can track any changes in S3

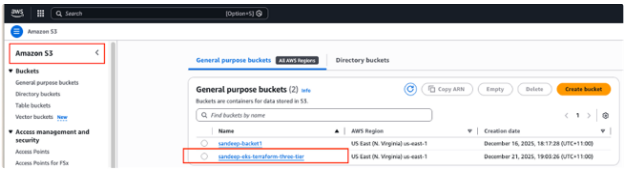
```
aws s3api put-bucket-versioning \
  --bucket sandeep-eks-terraform-three-tier \
  --versioning-configuration Status=Enabled
```

Enable Default Encryption

```
aws s3api put-bucket-encryption \
  --bucket sandeep-eks-terraform-three-tier \
  --server-side-encryption-configuration '{
    "Rules": [{
      "ApplyServerSideEncryptionByDefault": {
        "SSEAlgorithm": "AES256"
      }
    }]
  }'
```



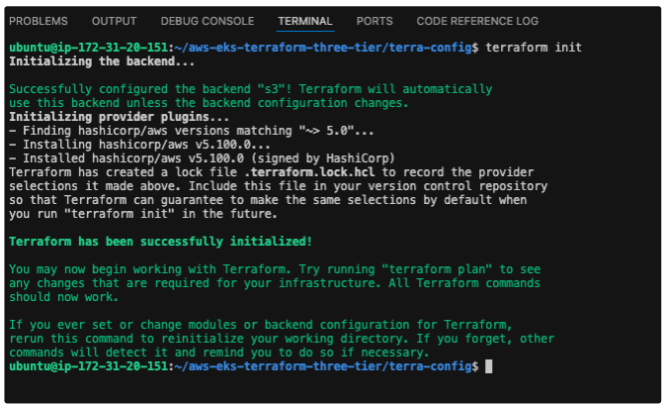
Lets validate the same in AWS Console by going to AWS and selecting S3



Step 5 : Provision the files with Terraform

Since the Terraform files are sitting within the backend, we will now move to the directory where the Terraform files are sitting

We will now initiate the Terraform backend



We will now check to see if the terraform plan to validate the configuration,Now we're ready to provision the infrastructure! Run the following command:

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

Understanding the Terraform Code Structure

Option One : Broken down each file into a module to understand

I've broken the configuration into logical modules for clarity and scalability. Here's a quick overview:

- provider.tf: Specifies the cloud provider. In our case, it's AWS (no surprise there!).
- backend.tf: Configures Terraform to store state remotely in our S3 bucket.
- ecr.tf: Creates two public repositories in ECR: 3-tier-frontend and 3-tier-backend for storing Docker images.
- vpc.tf: Fetches the default VPC and subnet details.
- role.tf: Defines IAM roles:

§ One for the EKS cluster (includes AmazonEKSClusterPolicy)

§ One for the Node Group (includes policies like AmazonEKSWorkerNodePolicy, AmazonEC2ContainerRegistryReadOnly, and AmazonEKS_CNI_Policy)

- eks.tf: Provisions the EKS cluster named Three-tier-cloud.
- node_group.tf: Creates the worker node group for the cluster with one t2.medium EC2 instance.

Option Two: A Main Terraform File for us to ensure we can update everything easily

· main.tf

o specifies all of the above in a single file which is just an amalgamation of all the TF files.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  CODE REFERENCE LOG

  # aws_iam_role_policy_attachment.example-AmazonEKSWorkerNodePolicy will be created
  + resource "aws_iam_role_policy_attachment" "example-AmazonEKSWorkerNodePolicy" {
      + id         = (known after apply)
      + policy_arn = "arn:aws:iam::aws:policy/AmazonEKSWorkerNodePolicy"
      + role       = "eks-node-group-example"
    }

  # aws_iam_role_policy_attachment.example-AmazonEKS_CNI_Policy will be created
  + resource "aws_iam_role_policy_attachment" "example-AmazonEKS_CNI_Policy" {
      + id         = (known after apply)
      + policy_arn = "arn:aws:iam::aws:policy/AmazonEKS_CNI_Policy"
      + role       = "eks-node-group-example"
    }

  # aws_launch_template.eks_node_launch_template will be created
  + resource "aws_launch_template" "eks_node_launch_template" {
      + arn             = (known after apply)
      + default_version = (known after apply)
      + id              = (known after apply)
      + instance_type   = "t2.medium"
      + latest_version  = (known after apply)
      + name            = "Three-tier-cloud-node-template"
      + name_prefix     = (known after apply)
      + tags            = {
          + "Name" = "Three-tier-cloud-node-template"
        }
      + tags_all        = {
          + "Name" = "Three-tier-cloud-node-template"
        }

      + metadata_options {
          + http_endpoint               = "enabled"
          + http_protocol_ipv6          = (known after apply)
          + http_put_response_hop_limit = 2
          + http_tokens                 = "required"
          + instance_metadata_tags      = (known after apply)
        }
    }

Plan: 11 to add, 0 to change, 0 to destroy.
```

We will now apply the Terraform Configuration

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

Once done it will show like this

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  CODE REFERENCE LOG

aws_eks_cluster.eks_cluster: Still creating... [06m30s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [06m40s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [06m50s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [07m00s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [07m10s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [07m20s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [07m30s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [07m40s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [07m50s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [08m00s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [08m10s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [08m20s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [08m30s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [08m40s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [08m50s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [09m00s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [09m10s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [09m20s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [09m30s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [09m40s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [09m50s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [10m00s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [10m10s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [10m20s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [10m30s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [10m40s elapsed]
aws_eks_cluster.eks_cluster: Still creating... [10m50s elapsed]
aws_eks_cluster.eks_cluster: Creation complete after 10m54s [id=three-tier-cloud-eks]
aws_launch_template.eks_node_launch_template: Creating...
aws_launch_template.eks_node_launch_template: Creation complete after 6s [id=lt-091be702f7998b1f0]
aws_eks_node_group.example: Creating...
aws_eks_node_group.example: Still creating... [00m10s elapsed]
aws_eks_node_group.example: Still creating... [00m20s elapsed]
aws_eks_node_group.example: Still creating... [00m30s elapsed]
aws_eks_node_group.example: Still creating... [00m40s elapsed]
aws_eks_node_group.example: Still creating... [00m50s elapsed]
aws_eks_node_group.example: Still creating... [01m00s elapsed]
aws_eks_node_group.example: Still creating... [01m10s elapsed]
aws_eks_node_group.example: Still creating... [01m20s elapsed]
aws_eks_node_group.example: Still creating... [01m30s elapsed]
aws_eks_node_group.example: Still creating... [01m40s elapsed]
aws_eks_node_group.example: Creation complete after 1m46s [id=three-tier-cloud-eks:eks-node-group]

Apply complete! Resources: 11 added, 0 changed, 0 destroyed.
ubuntu@ip-172-31-18-49:~/aws-eks-terraform-three-tier/Main-TF$
```

Step 6: Pushing the images to ECR

Lets fo to ECR on AWS Console

We will need to get the push command for Docker – for this we need to click on **three-tier-backend** then click on **View push commands**

```
aws ecr-public get-login-password --region us-east-1 | docker login --
username AWS --password-stdin public.ecr.aws/p2s5y711
```

We will now need to get back to the terminal and navigate to the directory for backend and use this push command to push the Docker Image to ECR repository.

Note 2: Enable buildkit issue : If you get an error message which wont allow you to connect.

You can use the following commands

Add current user to docker group

```
sudo usermod -aG docker $USER
```

Apply group changes

```
newgrp docker
```

Verify

```
docker ps
```

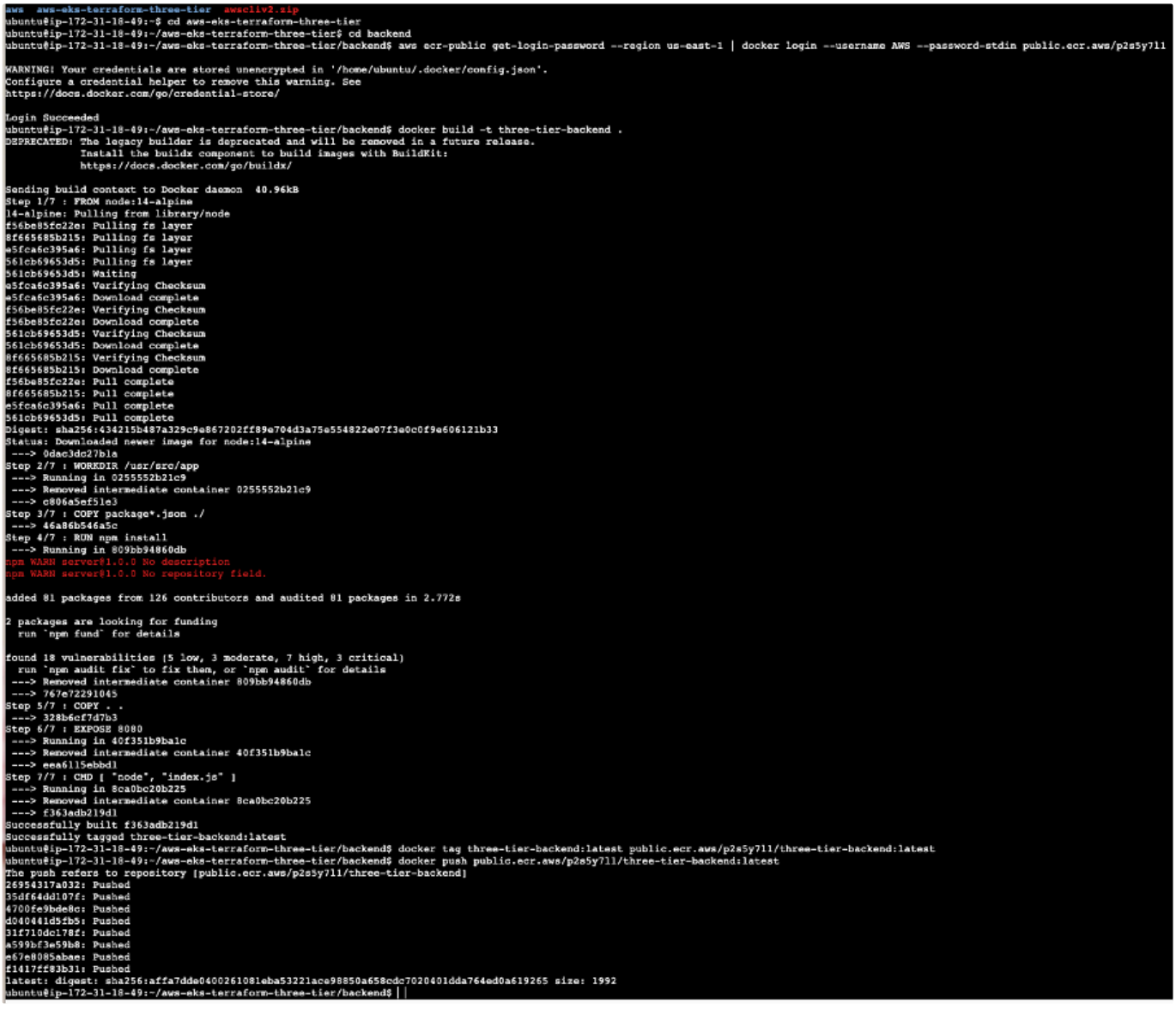
Temporary workaround(not recommended long term)

```
sudo docker build -t three-tier-backend .
```

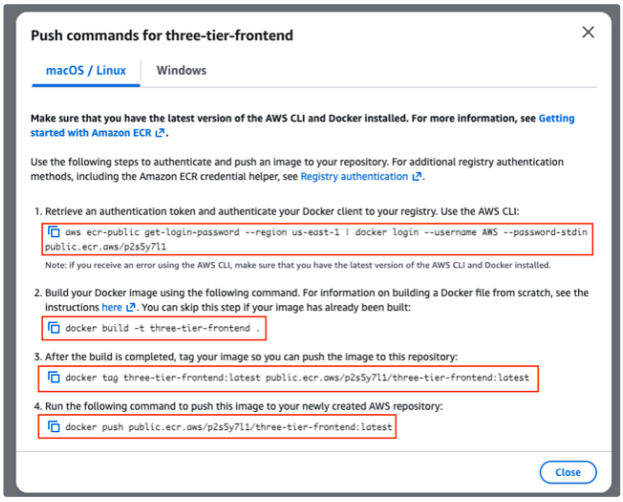
Best Option

```
DOCKER_BUILDKIT=1 docker build -t three-tier-backend .
```

We will now perform the same for the Frontend Functions as well



Navigating to ECR and selecting **three-tier-frontend** and then selecting **View push commands**



Once done you can validate the images by going to the ECR folders and viewing the images there

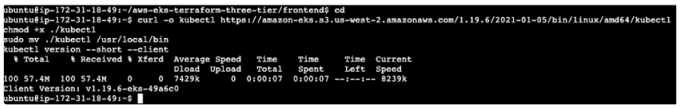
three-tier-backend

three-tier-frontend

Step 7 : Deploy to EKS with kubectl and Set Up Ingress via ALB

Now that your EKS cluster and ECR repositories are ready, it's time to interact with the cluster, deploy your workloads, and expose your application to the internet. We'll use **kubectl** for that — the command-line tool to manage Kubernetes clusters.

```
curl -o kubectl https://amazon-eks.s3.us-west-2.amazonaws.com/1.19.6/2021-01-05/bin/linux/amd64/kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin
kubectl version --short --client
```

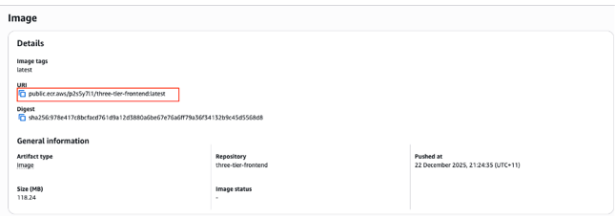


We will now need to install kubectl on our machine and deploy the Kubernetes manifests

Before we can do this – we must edit the Deployment.YAML file for both backend and frontend.

To do this – we must go to the folder containing the YAML file

```
vim frontend-deployment.yaml
```



You will see the image like this and we need to edit it with the unique characters highlighted above.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  namespace: workshop
  labels:
    role: frontend
    env: demo
spec:
  replicas: 1
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 25%
  selector:
    matchLabels:
      role: frontend
  template:
    metadata:
      labels:
        role: frontend
    spec:
      containers:
      - name: frontend
        image: public.ecr.aws/p2s5y7l1/three-tier-frontend:latest
        imagePullPolicy: Always
        env:
          - name: REACT_APP_BACKEND_URL
            value: "backend:8080" # assuming backend service is on port 8080
        ports:
        - containerPort: 3000
```

We will now do the same for backend as well

```
vim backend-deployment.yaml
```


```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
  namespace: workshop
  labels:
    role: backend
    env: demo
spec:
  replicas: 1
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 25%
  selector:
    matchLabels:
      role: backend
  template:
    metadata:
      labels:
        role: backend
    spec:
      containers:
      - name: backend
        image: public.ecr.aws/p2s5y7l1/three-tier-backend:latest
        imagePullPolicy: Always
        env:
          - name: MONGO_CONN_STR
            value: mongodb://mongodb-svc:27017/todo?directConnection=true
          - name: MONGO_USERNAME
            valueFrom:
              secretKeyRef:
                name: mongo-sec
                key: username
          - name: MONGO_PASSWORD
            valueFrom:
              secretKeyRef:
                name: mongo-sec
                key: password
        ports:
        - containerPort: 8080
        livenessProbe:
          httpGet:
            path: /ok
            port: 8080
          initialDelaySeconds: 2
          periodSeconds: 5
        readinessProbe:
          httpGet:
            path: /ok
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 5
          successThreshold: 1
```

Connect kubectl to Your EKS Cluster

Now configure kubectl to use your EKS cluster:

Steps followed are :

List clusters

```
aws eks list-clusters
```

Describe clusters

```
aws eks describe-cluster --name three-tier-cloud-eks --region us-east-1
```

We will now install eksctl

```
curl --silent --location
"https://github.com/weaveworks/eksctl/releases/latest/download/eksctl_$
(uname -s)_amd64.tar.gz" | tar xz -C /tmp
sudo mv /tmp/eksctl /usr/local/bin
eksctl version
```

```
ubuntu@ip-172-31-24-16:~/aws-eks-terraform-three-tier/k8s_manifests$ curl --silent --location "https://github.com/weaveworks/eksctl
/releases/latest/download/eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp
sudo mv /tmp/eksctl /usr/local/bin
eksctl version
0.221.0
```

We will now connect our ECS Clusters and connect kubeconfig

```
aws eks update-kubeconfig --region us-east-1 --name three-tier-cloud-
eks
```

```
ubuntu@ip-172-31-24-16:~/aws-eks-terraform-three-tier/k8s_manifests$ aws eks update-kubeconfig --region us-east-1 --name three-tier
-cloud-eks
Updated context arn:aws:eks:us-east-1:626635424094:cluster/three-tier-cloud-eks in /home/ubuntu/.kube/config
ubuntu@ip-172-31-24-16:~/aws-eks-terraform-three-tier/k8s_manifests$
```

Check the current context

```
kubectl config current-context
```

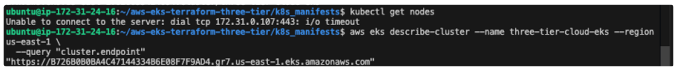
You should see

```
ubuntu@ip-172-31-24-16:~/aws-eks-terraform-three-tier/k8s_manifests$ kubectl config current-context
arn:aws:eks:us-east-1:626635424094:cluster/three-tier-cloud-eks
```

Now we will verify cluster nodes

```
kubectl get nodes
```

Tip : If kubectl get nodes returns an error message like the below



And after navigating to the page you see this error message

If it returns a **public DNS**, check that your **security group allows port 443** from your IP

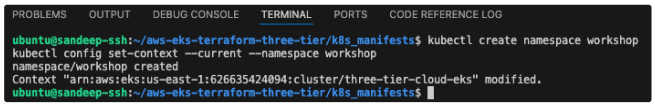
To do this – go to EKS→Networking→Add your IP or set to 0.0.0.0/0 to allow access from anywhere.

After fixing this – we should be able to see

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    CODE REFERENCE LOG

ubuntu@sandeep-ssh:~$ kubectl get nodes
NAME                          STATUS   ROLES    AGE   VERSION
ip-172-31-43-190.ec2.internal   Ready    <none>   33m   v1.34.2-eks-ecaa3a6
```

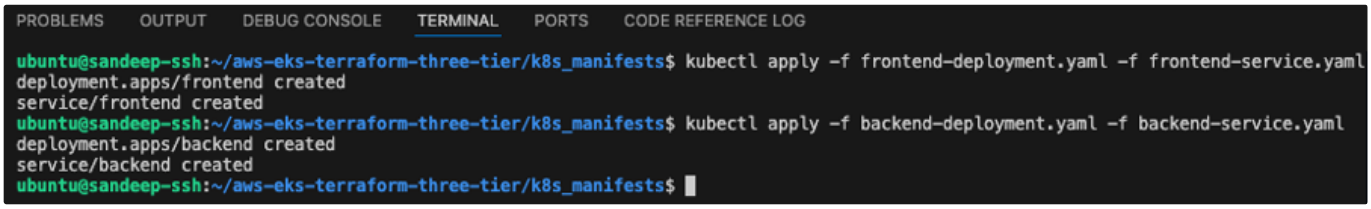
We will now create a namespace using the command

```
kubectl create namespace workshop
kubectl config set-context --current --namespace workshop
```



We will now deploy the updated manifests we updated earlier

```
kubectl apply -f frontend-deployment.yaml -f frontend-service.yaml
kubectl apply -f backend-deployment.yaml -f backend-service.yaml
```



We will now deploy MongoDB

```
cd mongo/
kubectl apply -f .
```

We will now check pods

```
kubectl get pods
```

Check services

```
kubectl get svc
```

Set Up Application Load Balancer (ALB) and Ingress

To route external traffic into your Kubernetes services, we'll use an **AWS Application Load Balancer** along with an **Ingress Controller**.

The IAM policy json is present inside the kubernetes manifests dir:

k8s_manifests/

Create the IAM policy in AWS:

```
aws iam create-policy \
  --policy-name AWSLoadBalancerControllerIAMPolicy \
  --policy-document file://iam_policy.json
```

We've already set up the OIDC provider earlier – we will now associate it

```
eksctl utils associate-iam-oidc-provider \
  --region=us-east-1 \
  --cluster=three-tier-cloud-eks \
  --approve
```

We will now create a service account for the load balancer – I've added my account number so if you wish to run for your project you can change it

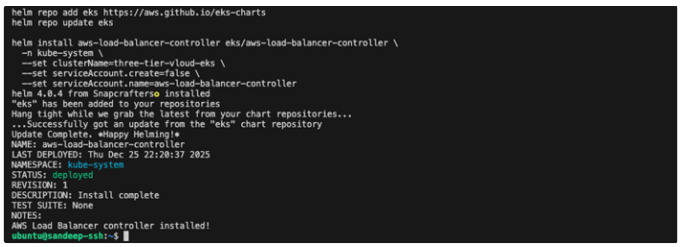
```
eksctl create iamserviceaccount \
  --cluster=three-tier-cloud-eks \
  --namespace=kube-system \
  --name=aws-load-balancer-controller \
  --role-name AmazonEKSLoadBalancerControllerRole \
  --attach-policy-
arn=arn:aws:iam::AWS_ACC_ID:policy/AWSLoadBalancerControllerIAMPolicy \
  --approve \
  --region=us-east-1
```

We will now install Helm and deploy the Load Balancer Controller

```
sudo snap install helm --classic

```

```
helm repo add eks https://aws.github.io/eks-charts
helm repo update eks

helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
  -n kube-system \
  --set clusterName=three-tier-cloud-eks \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller
```

Lets check if its running



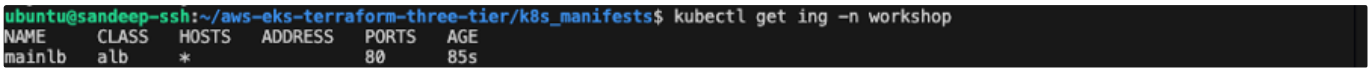

```
kubectl get deployment -n kube-system aws-load-balancer-controller
```

Now lets navigate to the k8s_manifests folder and apply the ingress

```
kubectl apply -f full_stack_lb.yaml
```

Wait for 5–7 minutes to allow the ingress and ALB to be fully provisioned

If not provisioned – it will show like this where there is no address showing up



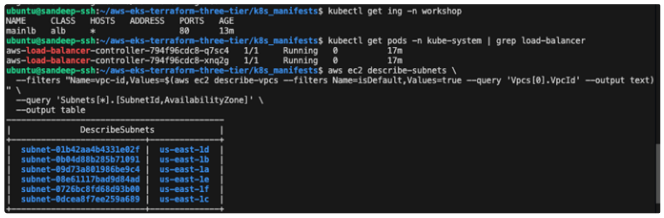
Access your application

```
kubectl get ing -n workshop
```

> *Tip*: ALB Address may take over 7-8 mins to show up

It was over 10 mins and the address didn't pop up so ran command to get the list of all subnets and it showed the below

```
aws ec2 describe-subnets \
  --filters "Name=vpc-id,Values=$(aws ec2 describe-vpcs --filters Name=isDefault,Values=true --query 'Vpcs[0].VpcId' --output text)" \
  --query 'Subnets[*].[SubnetId,AvailabilityZone]' \
  --output table
```



Ran the command to replace the public subnets to route the application

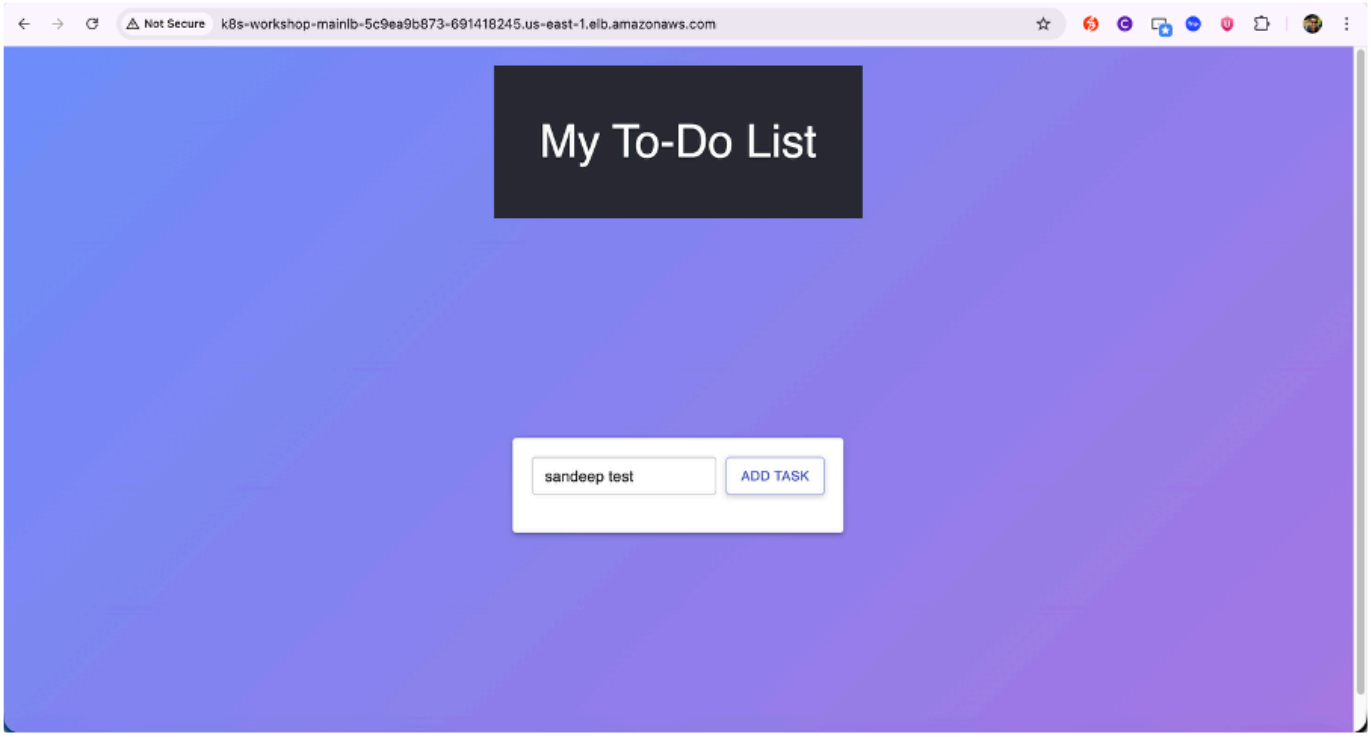
> AWS Load Balancer Controller discovers subnets using Kubernetes tags — without them, ALBs cannot be provisioned.

```
aws ec2 create-tags \
  --resources subnet-0b04d88b285b71091 subnet-01b42aa4b4331e02f \
  --tags \
    Key=kubernetes.io/cluster/three-tier-cloud-eks,Value=shared \
    Key=kubernetes.io/role/elb,Value=1
```

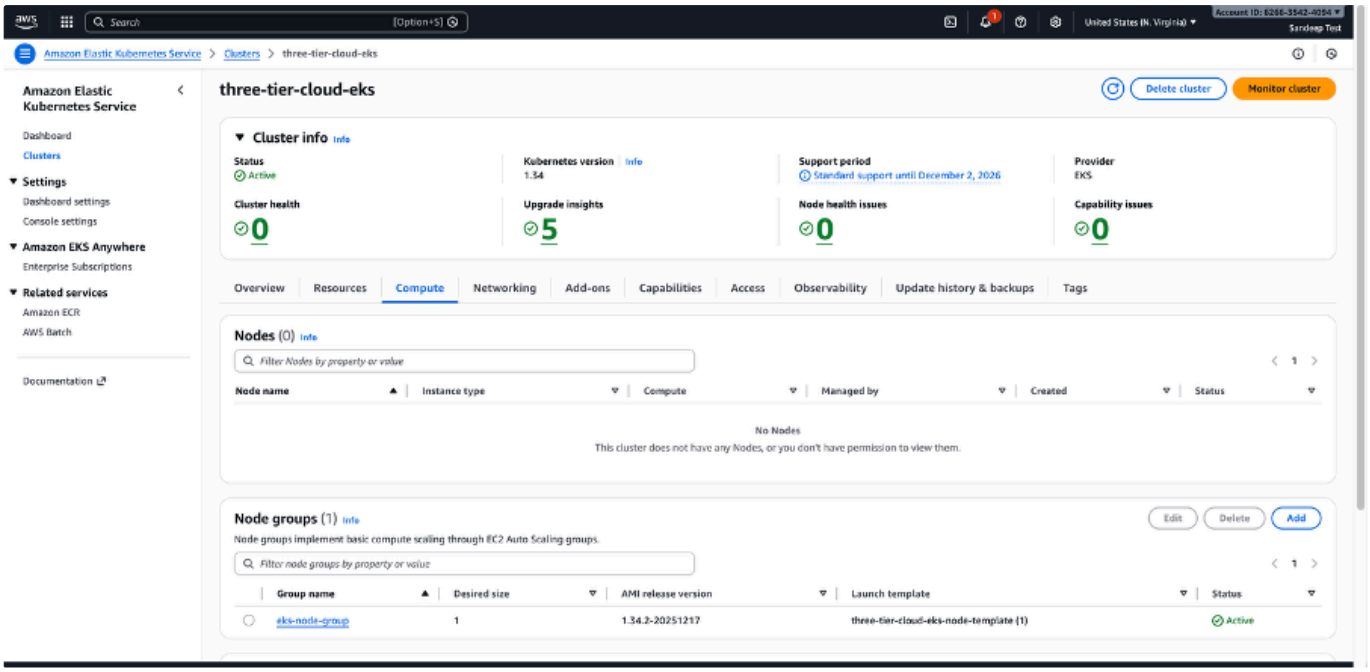
Re ran code

```
kubectl annotate serviceaccount aws-load-balancer-controller \
  -n kube-system \
  eks.amazonaws.com/role-arn=arn:aws:iam::626635424094:role/AmazonEKSLoadBalancerControllerRole

helm repo add eks https://aws.github.io/eks-charts
helm repo update

helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
  -n kube-system \
  --set clusterName=three-tier-cloud-eks \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller \
  --set region=us-east-1
```

This resolved the issue.



Screenshots from EKS

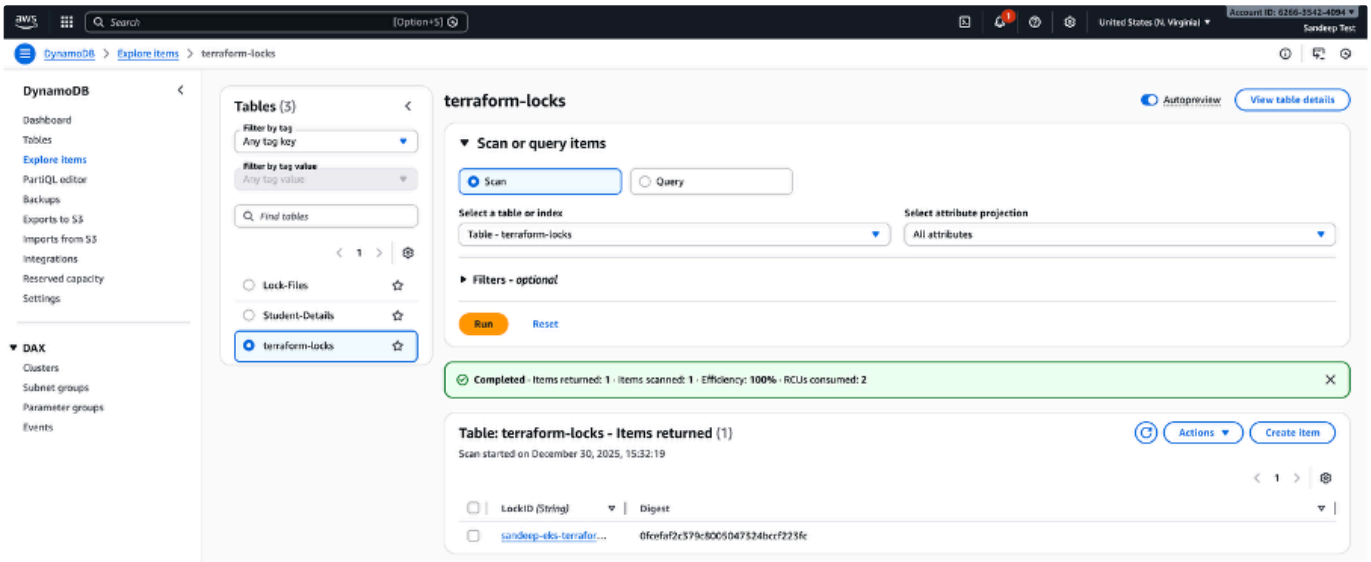


Elastic Container Registry - AWS ECR

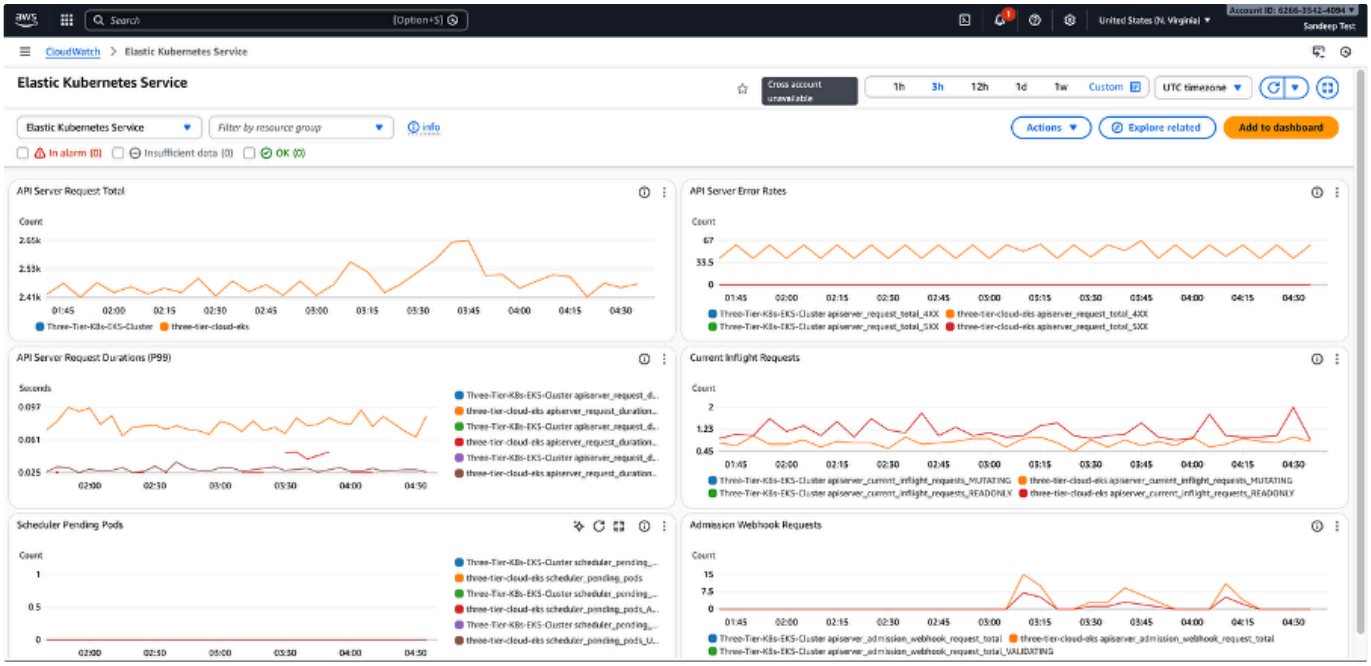
ECR Frontend

ECR Backend

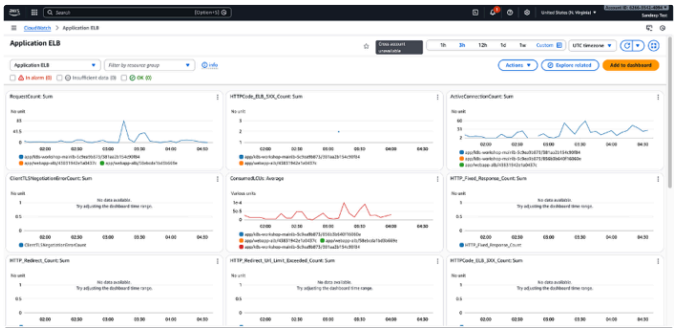
Dynamo DB



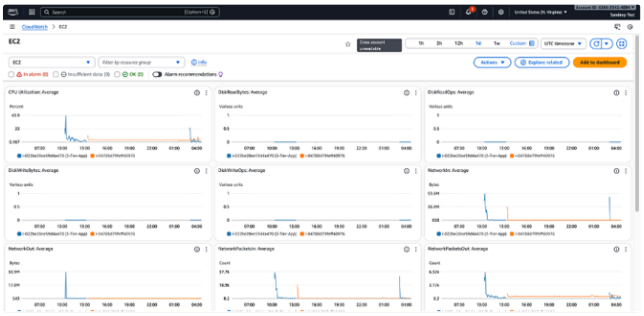
Cloudwatch-EKS



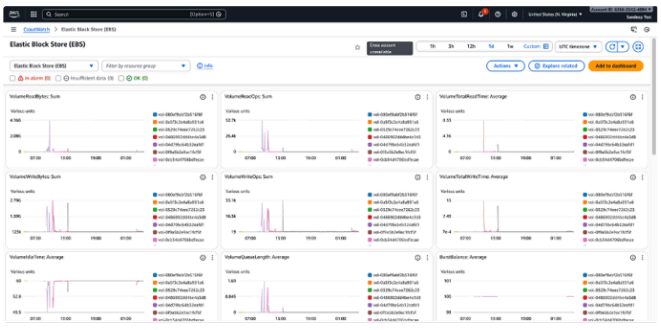
Cloudwatch -ELB



Cloudwatch-EC2



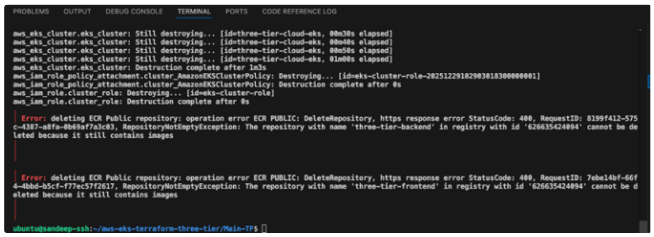
Cloudwatch - EBS



Step 10: Cleaning up AWS Resources

To avoid any un-necessary costs – we now need to go ahead and delete all our resources. Since we used Terraform – we can destroy/delete all resources with a touch of a single command.

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



We will need to go to ECR and delete the images from the AWS Console manually

---

Problem #	Issue Faced	Steps Taken to Resolve	Final Resolution
1	Backend targets unhealthy	Fixed health check path to `/backend`	Backend target became healthy
2	Frontend ImagePullBackOff	Pulled latest image, updated manifest	Frontend pod running successfully
3	Deployment selector immutable	Deleted old pods to allow new rollout	Deployment updated successfully
4	Ingress path routing	Configured ALB annotation `actions.backend-rewrite`	Path-based routing `/backend` works
5	Port conflicts on local dev	Stopped local service running on 8080	Backend accessible via cluster service

---

Future Improvements

- Enable **HTTPS** with ACM certificates on ALB.
- Implement **CI/CD pipeline** for automated EKS deployments.
- Add **Terraform modules** for reusability across projects.