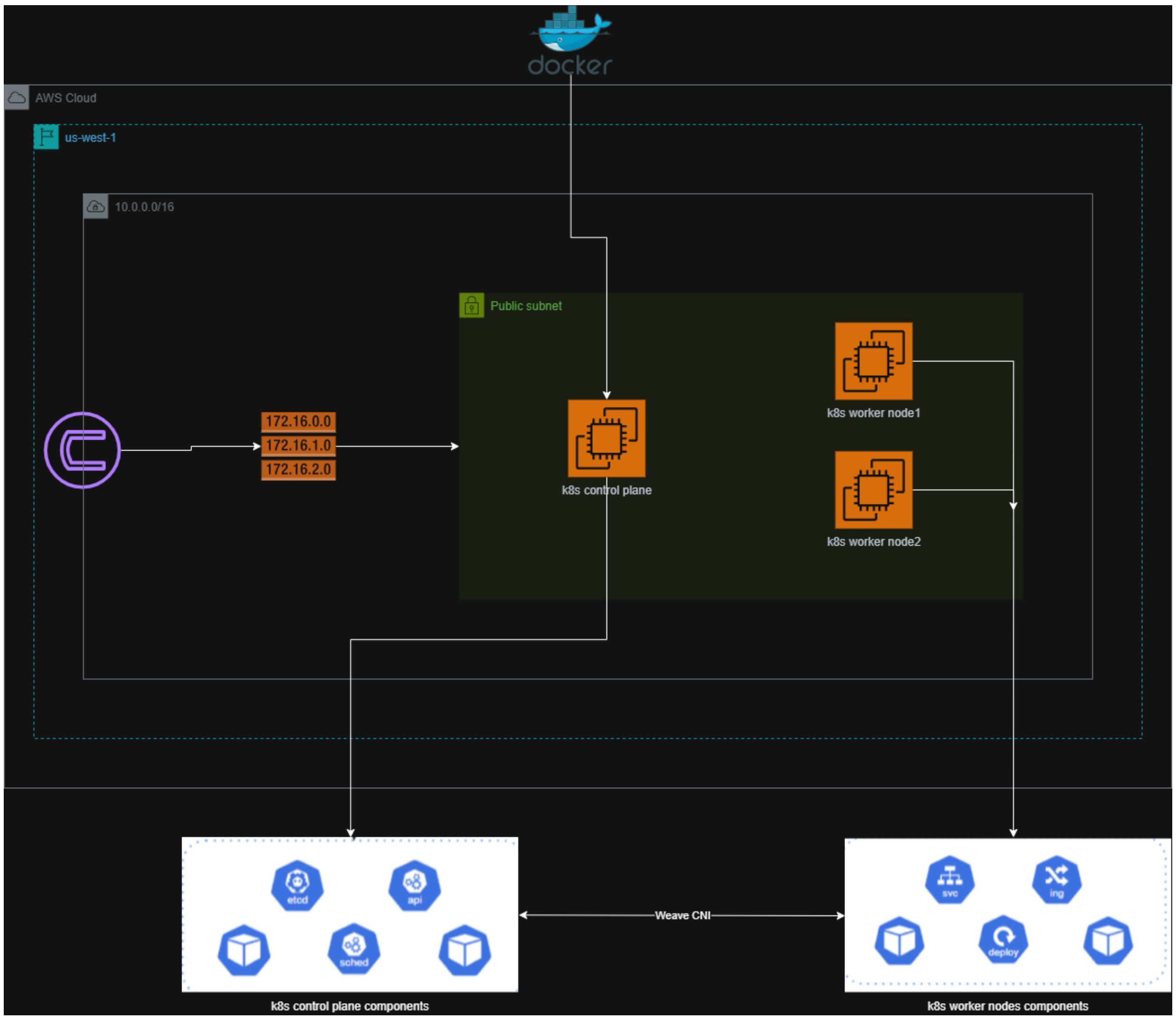
Deploying a Self-Managed Kubernetes Cluster on AWS EC2 with Node.js, NGINX Ingress, and OpenTelemetry Collector

Table of Contents

1) Executive summary

This project provisions a Kubernetes cluster on AWS EC2 using kubeadm. A custom Node.js UI application is deployed as a Kubernetes Deployment and exposed externally through an NGINX Ingress Controller running on a NodePort. OpenTelemetry instrumentation in the app exports traces via OTLP to an OpenTelemetry Collector deployed in-cluster. Collector logs verify that traces and spans are successfully received.

2) Architecture Diagram



3.0 Components

This section outlines the core infrastructure and application components used in the Kubernetes-based solution.

3.1 EC2 Control Plane Node (k8s-cp)

A dedicated EC2 instance acts as the Kubernetes control plane, provisioned using **kubeadm**. It runs the core control plane services including the API server, scheduler, controller manager, and etcd. The control plane manages cluster state, scheduling, and provides administrative access via kubectl.

3.2 EC2 Worker Nodes (k8s-w1, k8s-w2)

Two EC2 instances are configured as worker nodes and joined to the cluster using kubeadm join. These nodes run application workloads, including the Node.js UI application and the OpenTelemetry Collector. Each worker node runs kubelet and kube-proxy to manage pods and service networking.

3.3 Container Network Interface (CNI) – Weave Net

Weave Net is used as the CNI plugin to provide pod-to-pod networking across all nodes. It enables seamless communication between pods on different EC2 instances without additional network configuration, making it well suited for learning and demo environments.

3.4 Node.js UI Application

The Node.js application is containerized and deployed as a Kubernetes Deployment with two replicas for basic availability. It serves a web UI and multiple API endpoints and is instrumented with OpenTelemetry auto-instrumentation to emit trace data. A ClusterIP Service exposes the application internally within the cluster.

3.5 NGINX Ingress Controller and Ingress Resource

An NGINX Ingress Controller is deployed to handle external HTTP traffic. It listens on a NodePort exposed by the worker nodes and routes incoming requests to internal services based on Ingress rules. A Kubernetes Ingress resource defines routing for / traffic to the Node.js application service.

3.6 OpenTelemetry Collector

The OpenTelemetry Collector runs as a Kubernetes Deployment and serves as a centralized trace receiver. It accepts OTLP trace data from the application, processes it using batch processors, and exports traces to the logging exporter. The collector is exposed internally via a ClusterIP Service and accessed using Kubernetes DNS.

4) AWS networking and EC2 provisioning

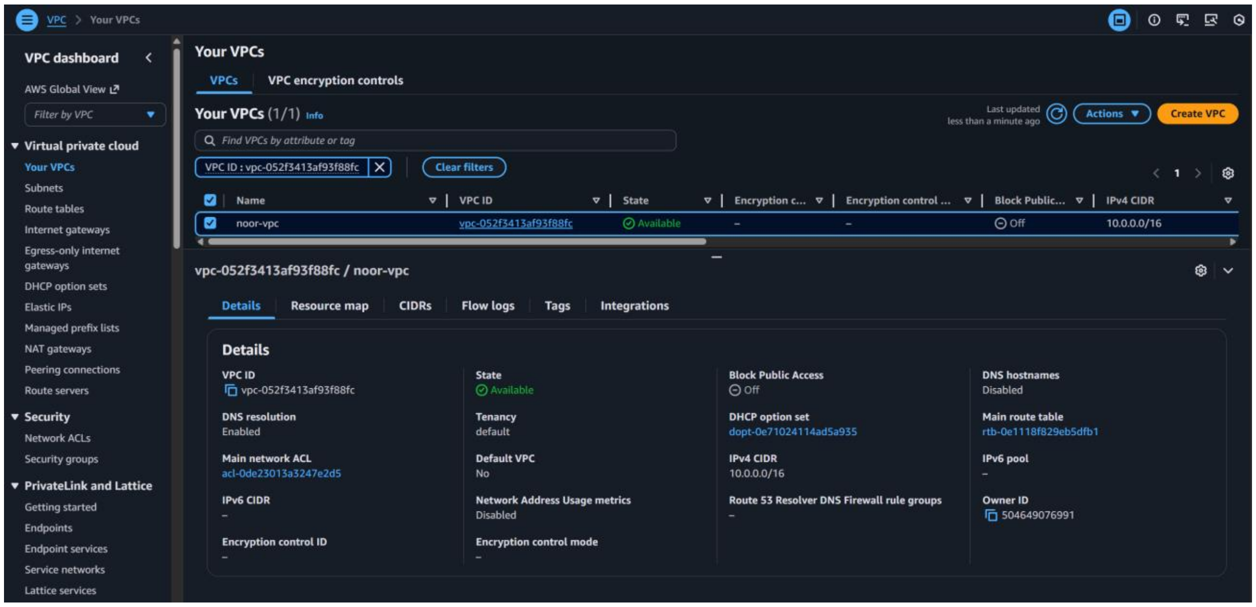
4.1 Virtual Private Cloud (VPC)

A dedicated Amazon VPC was created to isolate all Kubernetes-related resources.

VPC details:

- **Name:** noor-vpc
- **IPv4 CIDR block:** 10.0.0.0/16

The VPC provides a private IP address space large enough to accommodate control plane components, worker nodes, and pod networking.



Your VPCs

Your VPCs (1/1) Info

Last updated less than a minute ago

VPC ID : vpc-052f3413af93f88fc

Name	VPC ID	State	Encryption c...	Encryption control ...	Block Public...	IPv4 CIDR
noor-vpc	vpc-052f3413af93f88fc	Available	–	–	Off	10.0.0.0/16

vpc-052f3413af93f88fc / noor-vpc

Details

VPC ID vpc-052f3413af93f88fc	**State** Available	**Block Public Access** Off	**DNS hostnames** Disabled
DNS resolution Enabled	**Tenancy** default	**DHCP option set** dopt-0e71024114ad5a935	**Main route table** rtb-0e1118f829eb5dfb1
Main network ACL acl-0de23013a3247e2d5	**Default VPC** No	**IPv4 CIDR** 10.0.0.0/16	**IPv6 pool** –
IPv6 CIDR –	**Network Address Usage metrics** Disabled	**Route 53 Resolver DNS Firewall rule groups** –	**Owner ID** 504649076991
Encryption control ID –	**Encryption control mode** –		

4.2 Internet Gateway

An Internet Gateway was created and attached to the VPC to enable outbound internet access and inbound connectivity for SSH and application traffic.

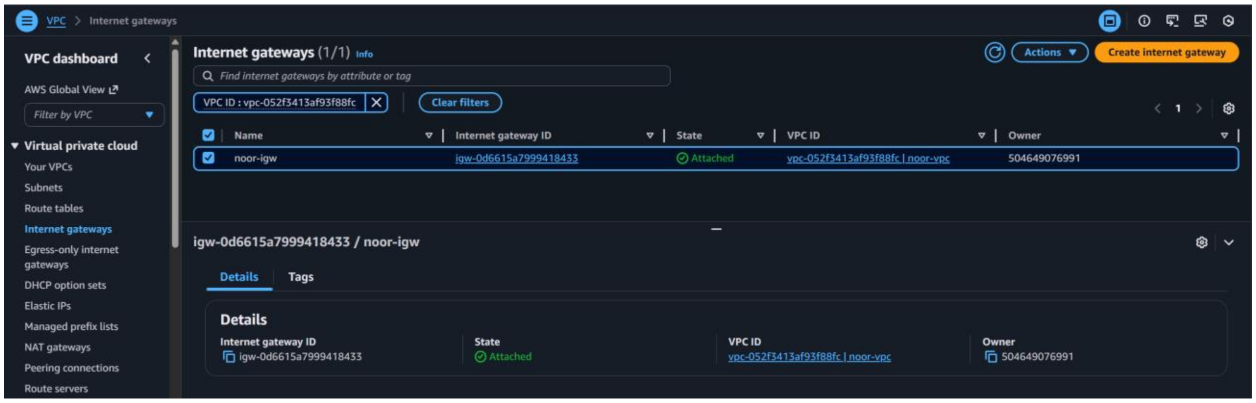
Internet Gateway details:

- **Name:** noor-igw
- **Attached to:** noor-vpc

The Internet Gateway allows EC2 instances in public subnets to:

- Pull container images

- Install packages during bootstrap
- Receive external HTTP traffic for the demo application


VPC > Internet gateways

VPC dashboard

AWS Global View

Virtual private cloud: Your VPCs, Subnets, Route tables, Internet gateways, Egress-only internet gateways, DHCP option sets, Elastic IPs, Managed prefix lists, NAT gateways, Peering connections, Route servers

Internet gateways (1/1) Info

Find internet gateways by attribute or tag

VPC ID : vpc-052f3413af93f88fc | Clear filters

Name	Internet gateway ID	State	VPC ID	Owner
noor-igw	igw-0d6615a7999418433	Attached	vpc-052f3413af93f88fc | noor-vpc	504649076991

igw-0d6615a7999418433 / noor-igw

Details | Tags

Details

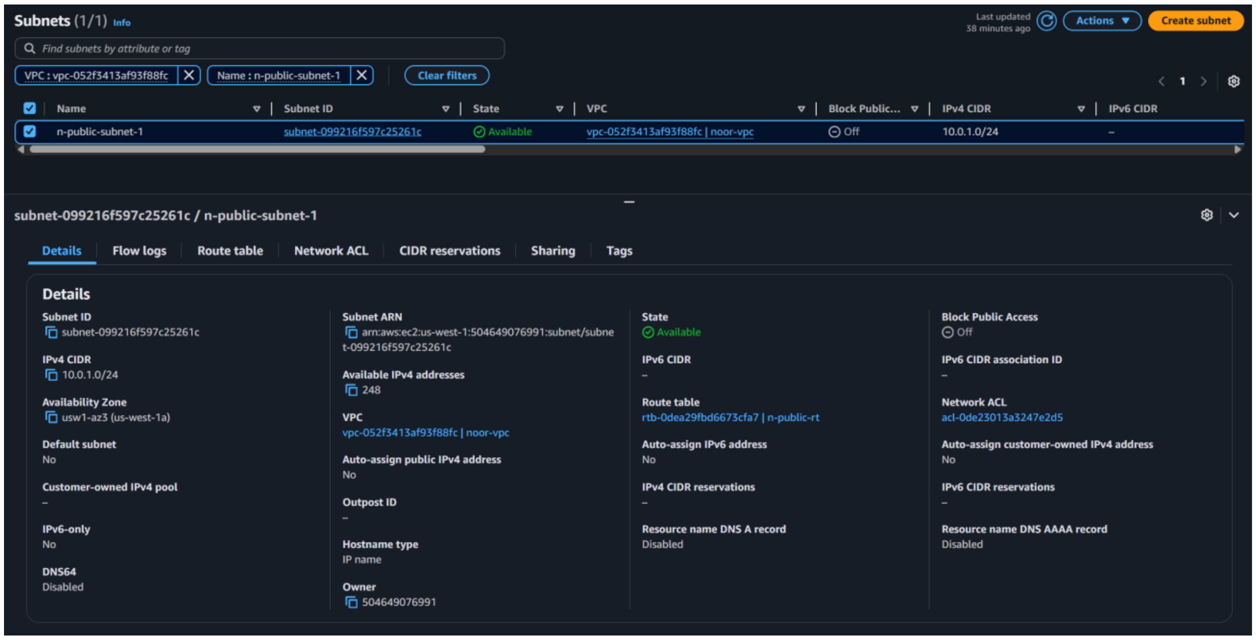
Internet gateway ID: igw-0d6615a7999418433
State: Attached
VPC ID: vpc-052f3413af93f88fc | noor-vpc
Owner: 504649076991

4.3 Subnet

For simplicity, the cluster uses a single **public subnet** located in one Availability Zone.

- **Availability Zone:** us-west-1a
- Public subnet with automatic public IP assignment
- Route table includes a default route (0.0.0.0/0) to the Internet Gateway

All EC2 instances (control plane and worker nodes) are deployed into this subnet. This design reduces complexity and makes external access straightforward for learning and testing.


Subnets (1/1) Info

Last updated 38 minutes ago

Find subnets by attribute or tag

VPC : vpc-052f3413af93f88fc | Name : n-public-subnet-1 | Clear filters

Name	Subnet ID	State	VPC	Block Public...	IPv4 CIDR	IPv6 CIDR
n-public-subnet-1	subnet-099216f597c25261c	Available	vpc-052f3413af93f88fc | noor-vpc	Off	10.0.1.0/24	–

subnet-099216f597c25261c / n-public-subnet-1

Details | Flow logs | Route table | Network ACL | CIDR reservations | Sharing | Tags

Details

Subnet ID: subnet-099216f597c25261c
IPv4 CIDR: 10.0.1.0/24
Availability Zone: usw1-az3 (us-west-1a)
Default subnet: No
Customer-owned IPv4 pool: –
IPv6-only: No
DNS64: Disabled

Subnet ARN: arn:aws:ec2:us-west-1:504649076991:subnet/subnet-099216f597c25261c
Available IPv4 addresses: 248
VPC: vpc-052f3413af93f88fc | noor-vpc
Auto-assign public IPv4 address: No
Outpost ID: –
Hostname type: IP name
Owner: 504649076991

State: Available
IPv6 CIDR: –
Route table: rtb-0dea29fbd6673cfa7 | n-public-rt
Auto-assign IPv6 address: No
IPv4 CIDR reservations: –
Resource name DNS A record: Disabled

Block Public Access: Off
IPv6 CIDR association ID: –
Network ACL: acl-0de23013a3247e2d5
Auto-assign customer-owned IPv4 address: No
IPv6 CIDR reservations: –
Resource name DNS AAAA record: Disabled

Note: In production environments, control plane and worker nodes should be deployed in private subnets across multiple Availability Zones with NAT gateways and load balancers.

4.4 Route Table

A public route table was configured to allow instances in the public subnets to communicate with the internet.

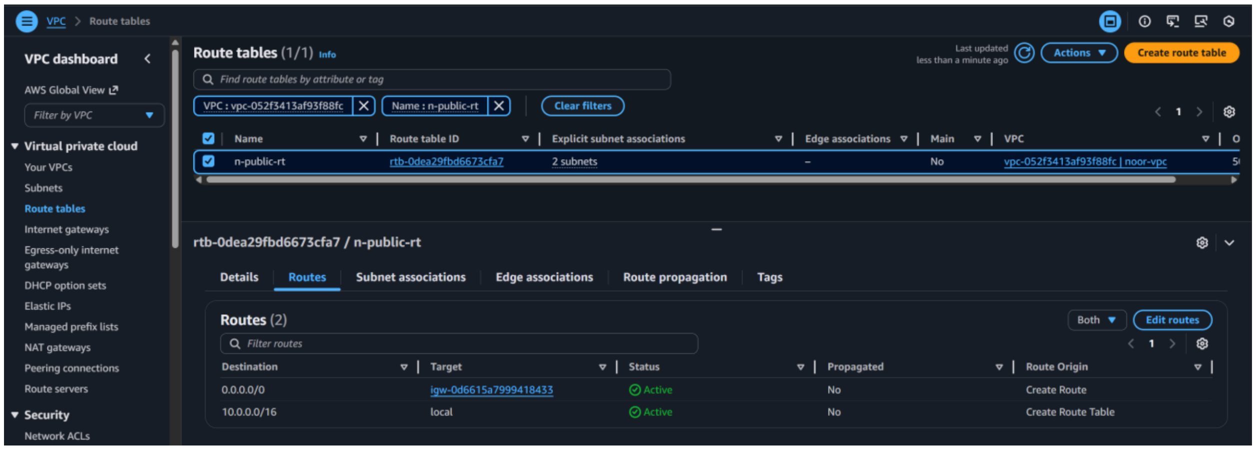
Route Table Details:

- **Name:** n-public-rt
- **VPC:** Associated with the project VPC (noor-vpc)

Configured Routes:

- **0.0.0.0/0 → Internet Gateway**
  Enables outbound internet access for resources in the associated public subnets.

- **10.0.0.0/16 → local**
  Default local route enabling internal communication within the VPC.



Route tables (1/1) Info

VPC : vpc-052f3413af93f88fc | Name : n-public-rt

Name	Route table ID	Explicit subnet associations	Edge associations	Main	VPC
n-public-rt	rtb-0dea29fbd6673cfa7	2 subnets	–	No	vpc-052f3413af93f88fc | noor-vpc

rtb-0dea29fbd6673cfa7 / n-public-rt

Routes (2)

Destination	Target	Status	Propagated	Route Origin
0.0.0.0/0	igw-0d6615a7999418433	Active	No	Create Route
10.0.0.0/16	local	Active	No	Create Route Table

4.5 Security Groups

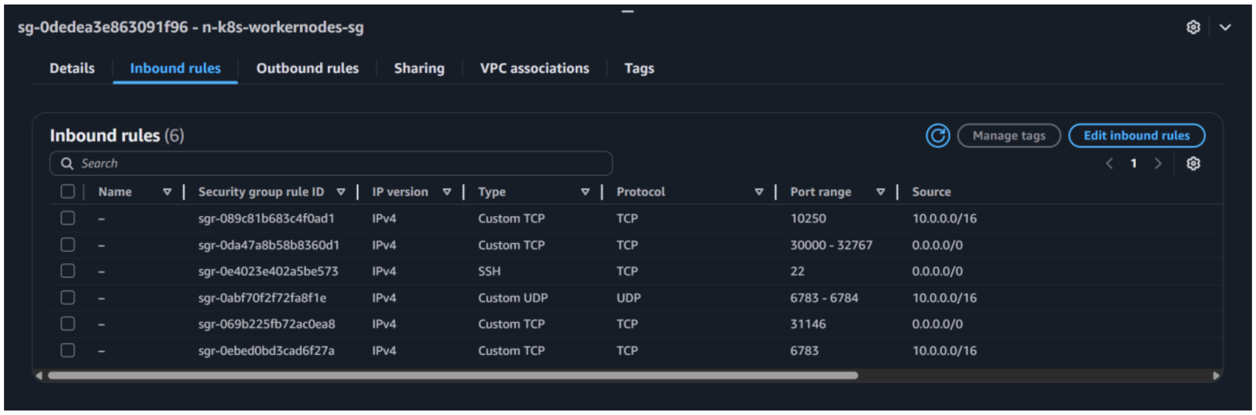
Separate security groups were created for the control plane and worker nodes to clearly define access boundaries and required ports.

4.5.1 Control Plane Security Group (n-k8s-control-plane-sg)

This security group allows administrative access and internal Kubernetes communication.

Inbound rules:

- **22/tcp** from **0.0.0.0/0**
  SSH access for administration (restricted to user IP in production)
- **6443/tcp** from **0.0.0.0/0**
  Kubernetes API server access
- **2379–2380/tcp** from **10.0.0.0/16**
  etcd client and peer communication
- **10250–10259/tcp** from **10.0.0.0/16**
  Kubelet and control-plane component communication
- **6783/tcp**, **6783–6784/udp** from **10.0.0.0/16**
  Weave Net CNI pod networking



Name	Security group rule ID	IP version	Type	Protocol	Port range	Source
–	sgr-089c81b683c4f0ad1	IPv4	Custom TCP	TCP	10250	10.0.0.0/16
–	sgr-0da47a8b58b8360d1	IPv4	Custom TCP	TCP	30000 - 32767	0.0.0.0/0
–	sgr-0e4023e402a5be573	IPv4	SSH	TCP	22	0.0.0.0/0
–	sgr-0abf70f2f72fa8f1e	IPv4	Custom UDP	UDP	6783 - 6784	10.0.0.0/16
–	sgr-069b225fb72ac0ea8	IPv4	Custom TCP	TCP	31146	0.0.0.0/0
–	sgr-0ebed0bd3cad6f27a	IPv4	Custom TCP	TCP	6783	10.0.0.0/16

This configuration enables the control plane to manage worker nodes and maintain cluster state.

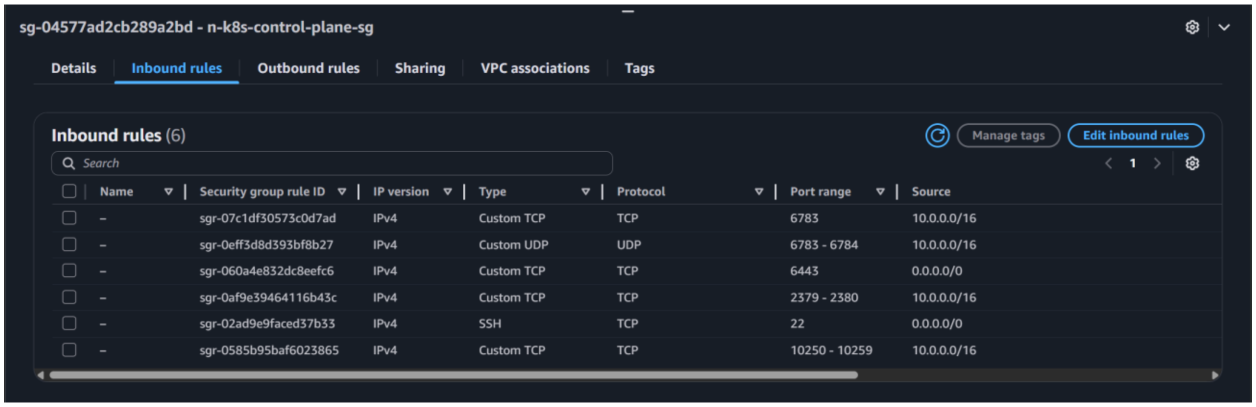
4.5.2 Worker Nodes Security Group (n-k8s-workernodes-sg)

This security group allows workload execution and external access via NodePort.

Inbound rules:

- **22/tcp** from **0.0.0.0/0**
  SSH access to worker nodes
- **10250/tcp** from **10.0.0.0/16**
  Kubelet communication

- **30000–32767/tcp** from **0.0.0.0/0**
  Kubernetes NodePort service range
- **31146/tcp** from **0.0.0.0/0**
  Specific NodePort used by the NGINX Ingress Controller
- **6783/tcp**, **6783–6784/udp** from **10.0.0.0/16**
  Weave Net CNI networking



sg-04577ad2cb289a2bd - n-k8s-control-plane-sg

Inbound rules (6)

Name	Security group rule ID	IP version	Type	Protocol	Port range	Source
–	sgr-07c1df30573c0d7ad	IPv4	Custom TCP	TCP	6783	10.0.0.0/16
–	sgr-0eff3d8d393bf8b27	IPv4	Custom UDP	UDP	6783 - 6784	10.0.0.0/16
–	sgr-060a4e832dc8eefc6	IPv4	Custom TCP	TCP	6443	0.0.0.0/0
–	sgr-0af9e39464116b43c	IPv4	Custom TCP	TCP	2379 - 2380	10.0.0.0/16
–	sgr-02ad9e9faced37b33	IPv4	SSH	TCP	22	0.0.0.0/0
–	sgr-0585b95baf6023865	IPv4	Custom TCP	TCP	10250 - 10259	10.0.0.0/16

This setup allows:

- External access to the application via Ingress
- Internal pod-to-pod communication
- Proper operation of Kubernetes networking components

4.6 EC2 Instance Configuration

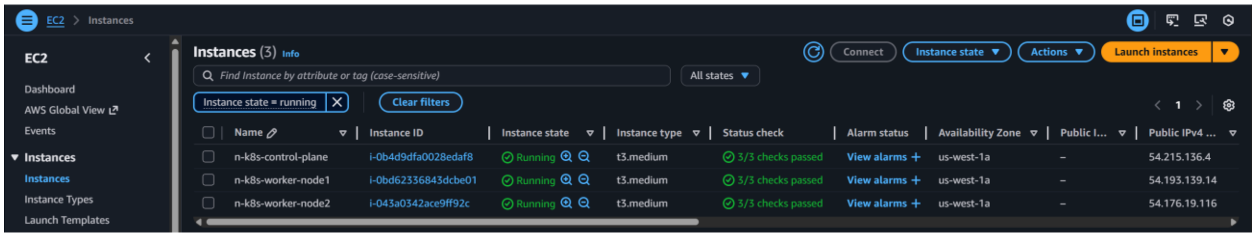
All EC2 instances in the cluster share a common baseline configuration to maintain consistency and simplify management. Differences between the control plane and worker nodes are limited to IAM instance profiles and assigned security groups.

Common Instance Configuration

The following settings are applied to all EC2 instances:

- **Operating System:** Ubuntu Linux 24.04
- **Instance type:** t3.medium
- **Key pair:** noorkey.pem
- **VPC:** noor-vpc
- **Subnet:** n-public-subnet-1 (us-west-1a)
- **Storage:** 8 GiB gp3 root volume

- **Public IP:** Enabled (via public subnet)



Instances (3) Info

Instance state = running

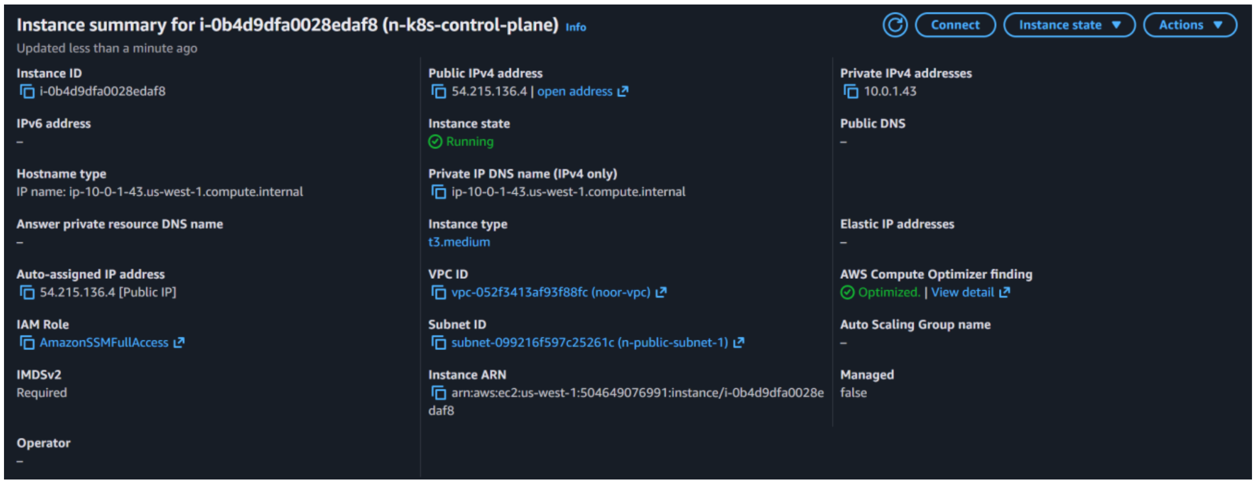
Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public I...	Public IPv4 ...
n-k8s-control-plane	i-0b4d9dfa0028edaf8	Running	t3.medium	3/3 checks passed	View alarms +	us-west-1a	–	54.215.136.4
n-k8s-worker-node1	i-0bd62336843dcbe01	Running	t3.medium	3/3 checks passed	View alarms +	us-west-1a	–	54.193.139.14
n-k8s-worker-node2	i-043a0342ace9ff92c	Running	t3.medium	3/3 checks passed	View alarms +	us-west-1a	–	54.176.19.116

This configuration provides sufficient CPU, memory, and storage resources for running the Kubernetes control plane components and application workloads in a learning environment.

Control Plane–Specific Configuration

The control plane instance uses the following role-specific settings:

- **Security group:** n-k8s-control-plane-sg
- **IAM instance profile:** AmazonSSMFullAccess



Instance summary for i-0b4d9dfa0028edaf8 (n-k8s-control-plane)

Updated less than a minute ago

Instance ID i-0b4d9dfa0028edaf8	**Public IPv4 address** 54.215.136.4 | open address	**Private IPv4 addresses** 10.0.1.43
IPv6 address –	**Instance state** Running	**Public DNS** –
Hostname type IP name: ip-10-0-1-43.us-west-1.compute.internal	**Private IP DNS name (IPv4 only)** ip-10-0-1-43.us-west-1.compute.internal	
Answer private resource DNS name –	**Instance type** t3.medium	**Elastic IP addresses** –
Auto-assigned IP address 54.215.136.4 [Public IP]	**VPC ID** vpc-052f3413af93f88fc (noor-vpc)	**AWS Compute Optimizer finding** Optimized. | View detail
IAM Role AmazonSSMFullAccess	**Subnet ID** subnet-099216f597c25261c (n-public-subnet-1)	**Auto Scaling Group name** –
IMDSv2 Required	**Instance ARN** arn:aws:ec2:us-west-1:504649076991:instance/i-0b4d9dfa0028edaf8	**Managed** false
Operator –		

The instance profile enables administrative access via AWS Systems Manager and supports secure remote management during cluster setup.

Worker Node–Specific Configuration

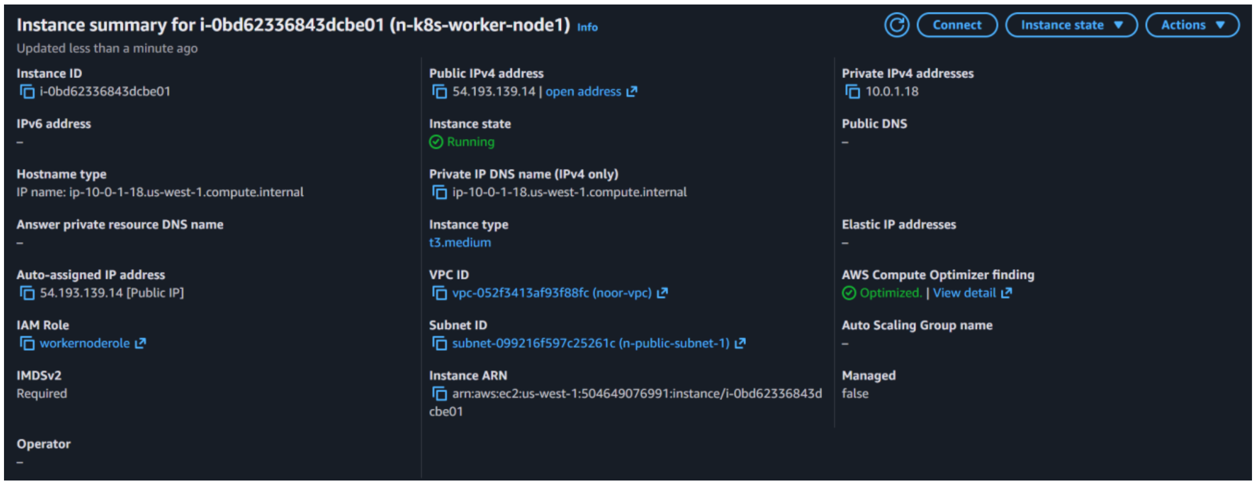
Both worker nodes share identical configurations with the following role-specific settings:

- **Security group:** n-k8s-workernodes-sg
- **IAM instance profile:** workernoderole, which includes:

- AmazonEC2ContainerRegistryReadOnly
- AmazonSSMManagedInstanceCore

These permissions allow worker nodes to pull container images from Amazon ECR (if required) and be managed using AWS Systems Manager.

Worker Node 1:

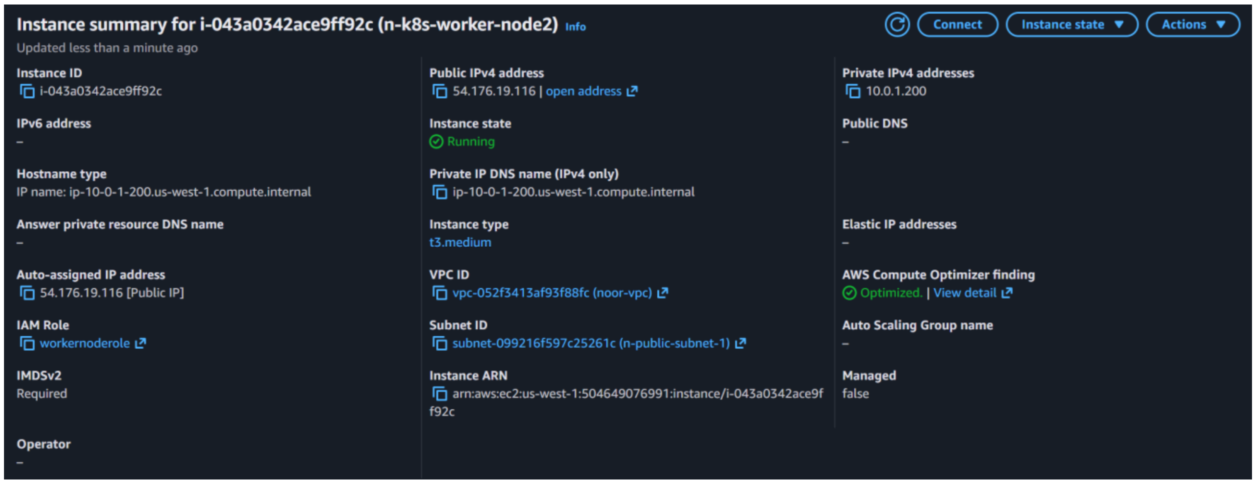


Instance summary for i-0bd62336843dcbe01 (n-k8s-worker-node1) Info

Updated less than a minute ago

Instance ID i-0bd62336843dcbe01	**Public IPv4 address** 54.193.139.14 | open address	**Private IPv4 addresses** 10.0.1.18
IPv6 address –	**Instance state** Running	**Public DNS** –
Hostname type IP name: ip-10-0-1-18.us-west-1.compute.internal	**Private IP DNS name (IPv4 only)** ip-10-0-1-18.us-west-1.compute.internal	
Answer private resource DNS name –	**Instance type** t3.medium	**Elastic IP addresses** –
Auto-assigned IP address 54.193.139.14 [Public IP]	**VPC ID** vpc-052f3413af93f88fc (noor-vpc)	**AWS Compute Optimizer finding** Optimized. | View detail
IAM Role workernoderole	**Subnet ID** subnet-099216f597c25261c (n-public-subnet-1)	**Auto Scaling Group name** –
IMDSv2 Required	**Instance ARN** arn:aws:ec2:us-west-1:504649076991:instance/i-0bd62336843dcbe01	**Managed** false
Operator –		

Worker Node 2:



Instance summary for i-043a0342ace9ff92c (n-k8s-worker-node2) Info

Updated less than a minute ago

Instance ID i-043a0342ace9ff92c	**Public IPv4 address** 54.176.19.116 | open address	**Private IPv4 addresses** 10.0.1.200
IPv6 address –	**Instance state** Running	**Public DNS** –
Hostname type IP name: ip-10-0-1-200.us-west-1.compute.internal	**Private IP DNS name (IPv4 only)** ip-10-0-1-200.us-west-1.compute.internal	
Answer private resource DNS name –	**Instance type** t3.medium	**Elastic IP addresses** –
Auto-assigned IP address 54.176.19.116 [Public IP]	**VPC ID** vpc-052f3413af93f88fc (noor-vpc)	**AWS Compute Optimizer finding** Optimized. | View detail
IAM Role workernoderole	**Subnet ID** subnet-099216f597c25261c (n-public-subnet-1)	**Auto Scaling Group name** –
IMDSv2 Required	**Instance ARN** arn:aws:ec2:us-west-1:504649076991:instance/i-043a0342ace9ff92c	**Managed** false
Operator –		

This standardized configuration ensures predictable behavior across all nodes while allowing role-based access control through security groups and IAM policies.

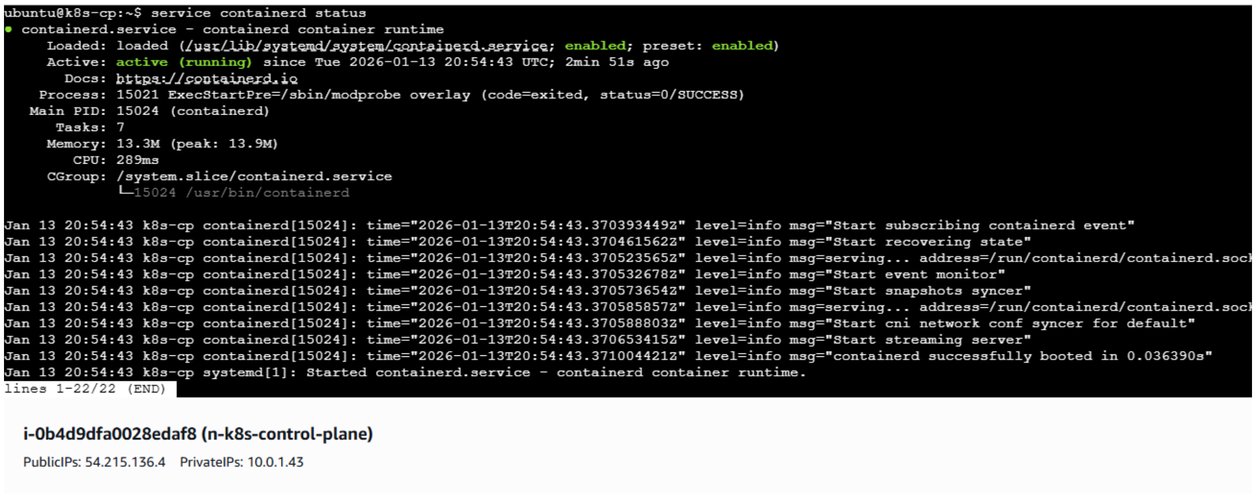
Production Note:
For real-world deployments, best practices include private subnets, restricted SSH access, security group references instead of CIDR blocks, and managed load balancers instead of NodePort exposure.

5. Kubernetes Installation on EC2 Using kubeadm

This section documents the installation and bootstrap of a Kubernetes cluster on Amazon EC2 instances using **kubeadm**. The setup follows standard Kubernetes installation practices and is based on the referenced guide, with minor adjustments during troubleshooting.

5.1 Verifying Container Runtime Installation (Control Plane)

The container runtime (**containerd**) was installed and verified on the control plane node. Kubernetes relies on a container runtime to manage containers, and containerd is a CNCF-supported runtime commonly used with kubeadm.



```
ubuntu@k8s-cp:~$ service containerd status
● containerd.service - containerd container runtime
     Loaded: loaded (/usr/lib/systemd/system/containerd.service; enabled; preset: enabled)
     Active: active (running) since Tue 2026-01-13 20:54:43 UTC; 2min 51s ago
       Docs: https://containerd.io
    Process: 15021 ExecStartPre=/sbin/modprobe overlay (code=exited, status=0/SUCCESS)
   Main PID: 15024 (containerd)
      Tasks: 7
     Memory: 13.3M (peak: 13.9M)
        CPU: 289ms
     CGroup: /system.slice/containerd.service
             └─15024 /usr/bin/containerd

Jan 13 20:54:43 k8s-cp containerd[15024]: time="2026-01-13T20:54:43.370393449Z" level=info msg="Start subscribing containerd event"
Jan 13 20:54:43 k8s-cp containerd[15024]: time="2026-01-13T20:54:43.370461562Z" level=info msg="Start recovering state"
Jan 13 20:54:43 k8s-cp containerd[15024]: time="2026-01-13T20:54:43.370523565Z" level=info msg=serving... address=/run/containerd/containerd.sock
Jan 13 20:54:43 k8s-cp containerd[15024]: time="2026-01-13T20:54:43.370532678Z" level=info msg="Start event monitor"
Jan 13 20:54:43 k8s-cp containerd[15024]: time="2026-01-13T20:54:43.370573654Z" level=info msg="Start snapshots syncer"
Jan 13 20:54:43 k8s-cp containerd[15024]: time="2026-01-13T20:54:43.370585857Z" level=info msg=serving... address=/run/containerd/containerd.sock
Jan 13 20:54:43 k8s-cp containerd[15024]: time="2026-01-13T20:54:43.370588803Z" level=info msg="Start cni network conf syncer for default"
Jan 13 20:54:43 k8s-cp containerd[15024]: time="2026-01-13T20:54:43.370653415Z" level=info msg="Start streaming server"
Jan 13 20:54:43 k8s-cp containerd[15024]: time="2026-01-13T20:54:43.371004421Z" level=info msg="containerd successfully booted in 0.036390s"
Jan 13 20:54:43 k8s-cp systemd[1]: Started containerd.service - containerd container runtime.
lines 1-22/22 (END)
```

i-0b4d9dfa0028edaf8 (n-k8s-control-plane)

PublicIPs: 54.215.136.4   PrivateIPs: 10.0.1.43

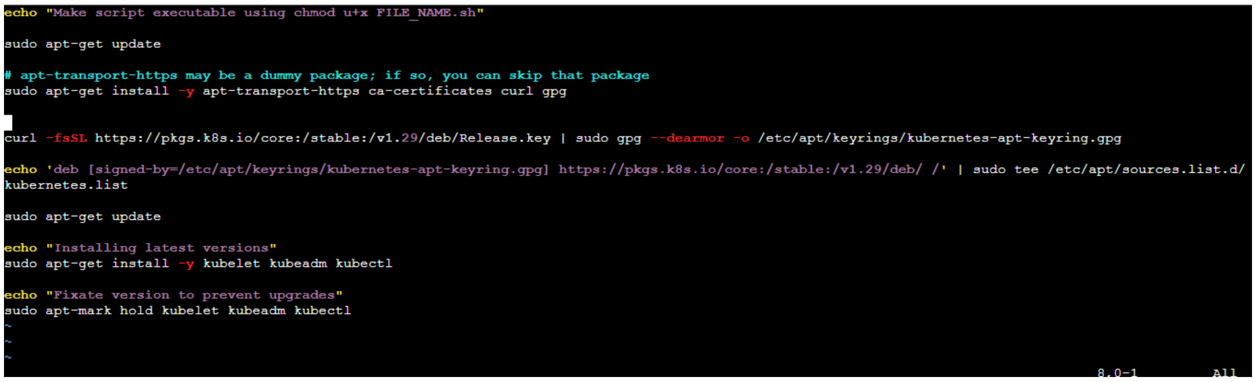
This confirms:

- containerd service is installed
- Service state is **active (running)**

This verification ensures the node is ready to host Kubernetes components.

5.2 Installing Kubernetes Packages (Control Plane)

The official Kubernetes APT repository was added to the system, followed by an update of package lists. This ensures that Kubernetes binaries are installed from a trusted and up-to-date source.


```
echo "Make script executable using chmod u+x FILE_NAME.sh"

sudo apt-get update

# apt-transport-https may be a dummy package; if so, you can skip that package
sudo apt-get install -y apt-transport-https ca-certificates curl gpg


curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.29/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg

echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /' | sudo tee /etc/apt/sources.list.d/
kubernetes.list

sudo apt-get update

echo "Installing latest versions"
sudo apt-get install -y kubelet kubeadm kubectl

echo "Fixate version to prevent upgrades"
sudo apt-mark hold kubelet kubeadm kubectl
~
~
~
                                                                                                          8,0-1         All
```

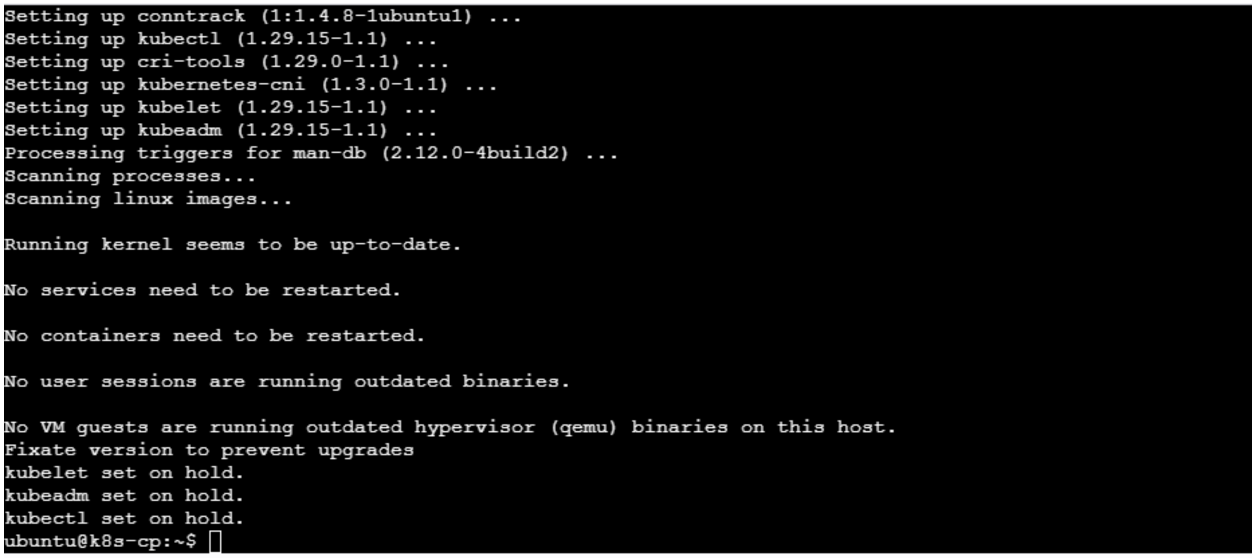
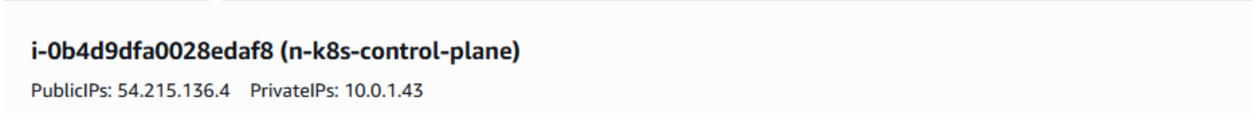
Commands performed:

- Added Kubernetes GPG signing key
- Configured Kubernetes APT repository
- Updated package index

5.3 Installing kubeadm, kubelet, and kubectl

The core Kubernetes tools were installed:

- **kubeadm** for cluster bootstrapping
- **kubelet** for node-level pod management
- **kubectl** for cluster administration

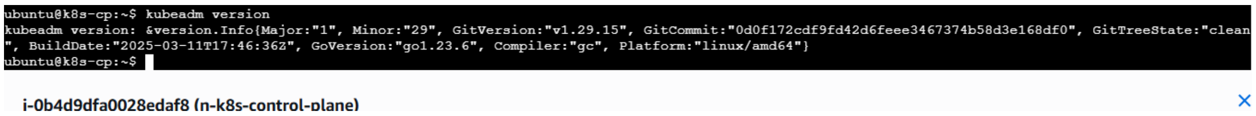

```
Setting up conntrack (1:1.4.8-1ubuntu1) ...
Setting up kubectl (1.29.15-1.1) ...
Setting up cri-tools (1.29.0-1.1) ...
Setting up kubernetes-cni (1.3.0-1.1) ...
Setting up kubelet (1.29.15-1.1) ...
Setting up kubeadm (1.29.15-1.1) ...
Processing triggers for man-db (2.12.0-4build2) ...
Scanning processes...
Scanning linux images...

Running kernel seems to be up-to-date.

No services need to be restarted.

No containers need to be restarted.

No user sessions are running outdated binaries.

No VM guests are running outdated hypervisor (qemu) binaries on this host.
Fixate version to prevent upgrades
kubelet set on hold.
kubeadm set on hold.
kubectl set on hold.
ubuntu@k8s-cp:~$
```

i-0b4d9dfa0028edaf8 (n-k8s-control-plane)

PublicIPs: 54.215.136.4   PrivateIPs: 10.0.1.43

The screenshot shows successful installation and that the packages were placed on hold to prevent unintended version upgrades.

5.4 Verifying Kubernetes Version

The installed Kubernetes version was verified using **kubeadm version** command. This confirms that the expected Kubernetes version (v1.29.15) is installed before initializing the cluster.

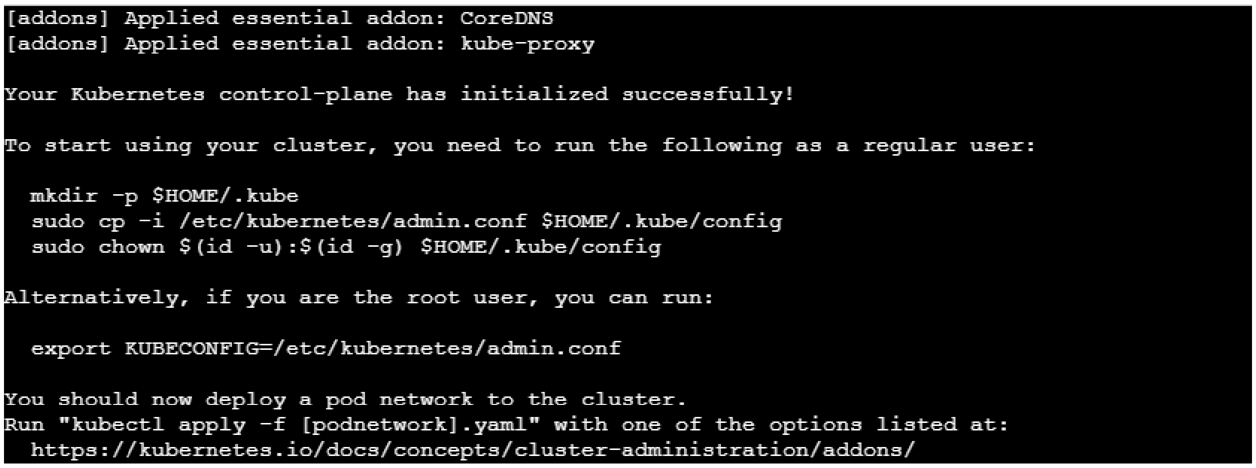


```
ubuntu@k8s-cp:~$ kubeadm version
kubeadm version: &version.Info{Major:"1", Minor:"29", GitVersion:"v1.29.15", GitCommit:"0d0f172cdf9fd42d6feee3467374b58d3e168df0", GitTreeState:"clean
", BuildDate:"2025-03-11T17:46:36Z", GoVersion:"go1.23.6", Compiler:"gc", Platform:"linux/amd64"}
ubuntu@k8s-cp:~$
```

5.5 Initializing the Kubernetes Control Plane

The Kubernetes control plane was initialized using **kubeadm init**.

- Bootstrapped control plane components
- Generated cluster certificates
- Deployed essential add-ons
- Produced the kubeadm join command for worker nodes

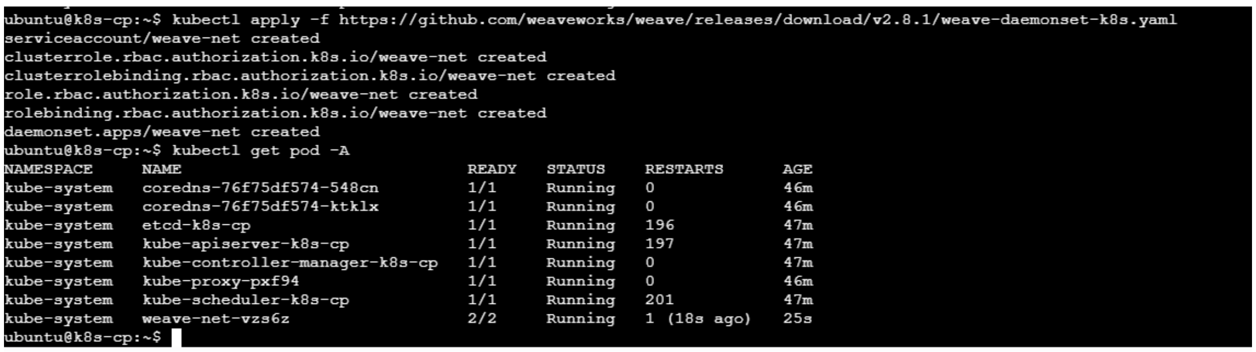


```
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/
```

The screenshot also shows configuration of the **kubeconfig** file for the local user, enabling **kubectl** access.

5.6 Installing the CNI Plugin (Weave Net)

Weave Net was installed as the Container Network Interface (CNI) plugin. The CNI is required for pod-to-pod networking and inter-node communication.



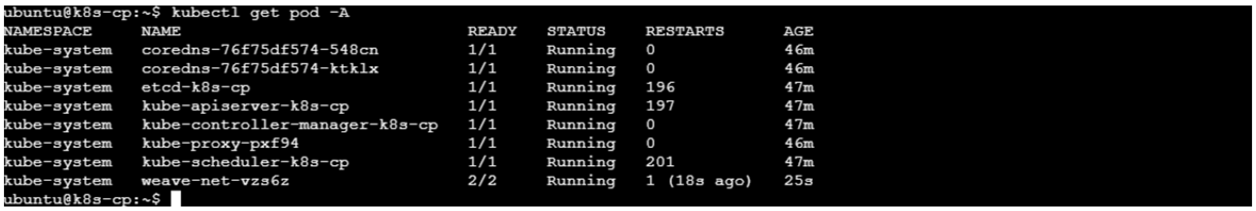
```
ubuntu@k8s-cp:~$ kubectl apply -f https://github.com/weaveworks/weave/releases/download/v2.8.1/weave-daemonset-k8s.yaml
serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
daemonset.apps/weave-net created
ubuntu@k8s-cp:~$ kubectl get pod -A
NAMESPACE     NAME                              READY   STATUS    RESTARTS      AGE
kube-system   coredns-76f75df574-548cn          1/1     Running   0             46m
kube-system   coredns-76f75df574-ktklx          1/1     Running   0             46m
kube-system   etcd-k8s-cp                       1/1     Running   196           47m
kube-system   kube-apiserver-k8s-cp             1/1     Running   197           47m
kube-system   kube-controller-manager-k8s-cp    1/1     Running   0             47m
kube-system   kube-proxy-pxf94                  1/1     Running   0             46m
kube-system   kube-scheduler-k8s-cp             1/1     Running   201           47m
kube-system   weave-net-vzs6z                   2/2     Running   1 (18s ago)   25s
ubuntu@k8s-cp:~$
```

This confirms that **Weave Net** resources including DaemonSet, roles, bindings etc. were created successfully

5.7 Verifying Core System Pods

After installing the control plane and CNI, system pods were verified across all namespaces. The output shows core Kubernetes components such as:

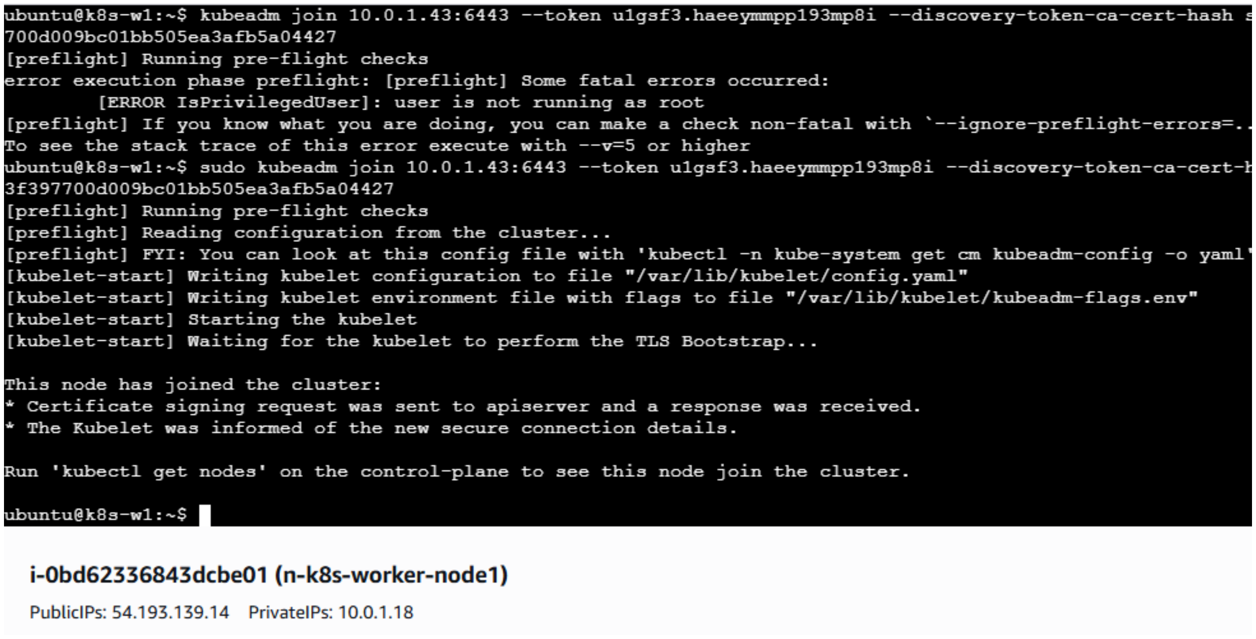
- kube-apiserver
- etcd
- kube-controller-manager
- kube-scheduler
- kube-proxy
- coredns
- weave-net



```
ubuntu@k8s-cp:~$ kubectl get pod -A
NAMESPACE     NAME                              READY   STATUS    RESTARTS      AGE
kube-system   coredns-76f75df574-548cn          1/1     Running   0             46m
kube-system   coredns-76f75df574-ktklx          1/1     Running   0             46m
kube-system   etcd-k8s-cp                       1/1     Running   196           47m
kube-system   kube-apiserver-k8s-cp             1/1     Running   197           47m
kube-system   kube-controller-manager-k8s-cp    1/1     Running   0             47m
kube-system   kube-proxy-pxf94                  1/1     Running   0             46m
kube-system   kube-scheduler-k8s-cp             1/1     Running   201           47m
kube-system   weave-net-vzs6z                   2/2     Running   1 (18s ago)   25s
ubuntu@k8s-cp:~$
```

This confirms that the control plane is operational.

5.8 Joining Worker Node 1 to the Cluster



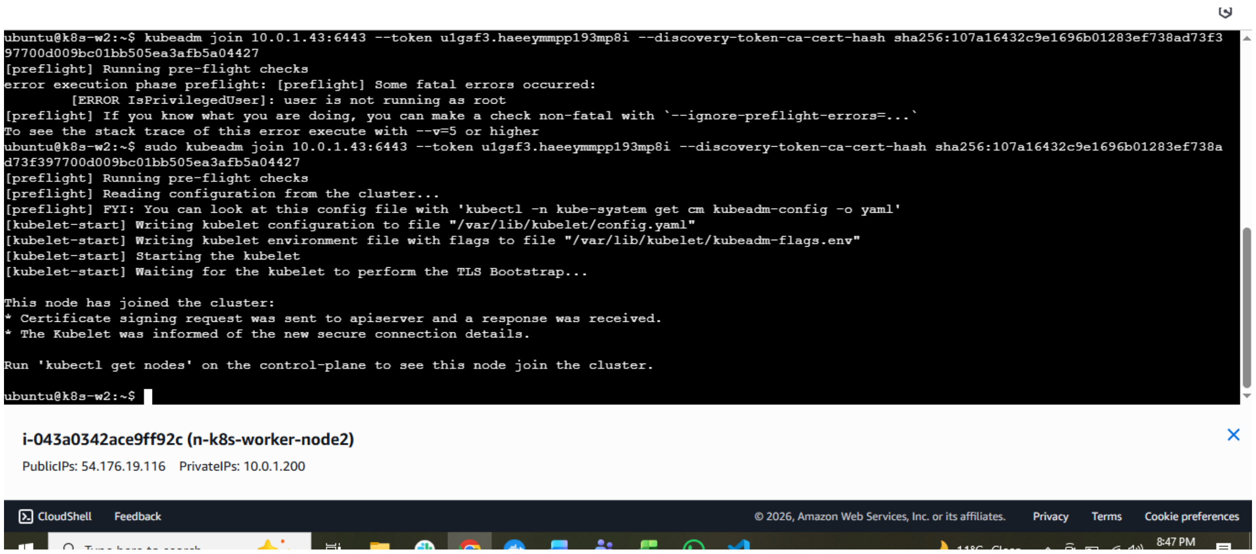
```
ubuntu@k8s-w1:~$ kubeadm join 10.0.1.43:6443 --token u1gsf3.haeeymmpp193mp8i --discovery-token-ca-cert-hash s
700d009bc01bb505ea3afb5a04427
[preflight] Running pre-flight checks
error execution phase preflight: [preflight] Some fatal errors occurred:
        [ERROR IsPrivilegedUser]: user is not running as root
[preflight] If you know what you are doing, you can make a check non-fatal with `--ignore-preflight-errors=..
To see the stack trace of this error execute with --v=5 or higher
ubuntu@k8s-w1:~$ sudo kubeadm join 10.0.1.43:6443 --token u1gsf3.haeeymmpp193mp8i --discovery-token-ca-cert-h
3f397700d009bc01bb505ea3afb5a04427
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

ubuntu@k8s-w1:~$
```

i-0bd62336843dcbe01 (n-k8s-worker-node1)

PublicIPs: 54.193.139.14   PrivateIPs: 10.0.1.18

The first worker node was joined to the cluster using the **kubeadm join command** generated during control plane initialization. The screenshot confirms successful execution of the join process.

5.9 Joining Worker Node 2 to the Cluster



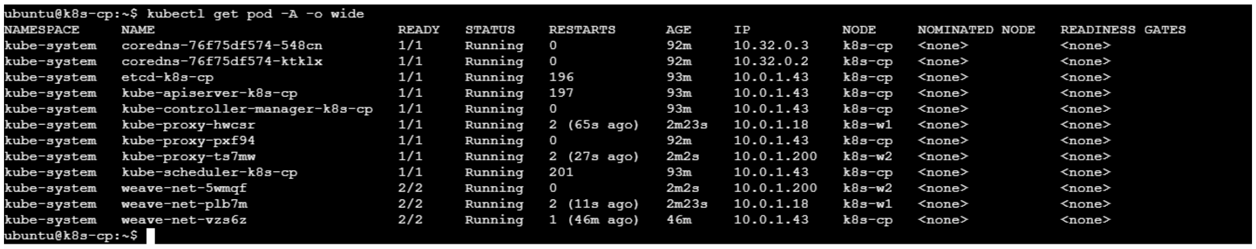
```
ubuntu@k8s-w2:~$ kubeadm join 10.0.1.43:6443 --token u1gsf3.haeeymmpp193mp8i --discovery-token-ca-cert-hash sha256:107a16432c9e1696b01283ef738ad73f3
97700d009bc01bb505ea3afb5a04427
[preflight] Running pre-flight checks
error execution phase preflight: [preflight] Some fatal errors occurred:
        [ERROR IsPrivilegedUser]: user is not running as root
[preflight] If you know what you are doing, you can make a check non-fatal with `--ignore-preflight-errors=...`
To see the stack trace of this error execute with --v=5 or higher
ubuntu@k8s-w2:~$ sudo kubeadm join 10.0.1.43:6443 --token u1gsf3.haeeymmpp193mp8i --discovery-token-ca-cert-hash sha256:107a16432c9e1696b01283ef738a
d73f397700d009bc01bb505ea3afb5a04427
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

ubuntu@k8s-w2:~$
```

i-043a0342ace9ff92c (n-k8s-worker-node2)

PublicIPs: 54.176.19.116   PrivateIPs: 10.0.1.200

The second worker node was joined using the same join command. This completes the initial cluster node registration process.

5.10 Verifying Cluster State After Worker Join



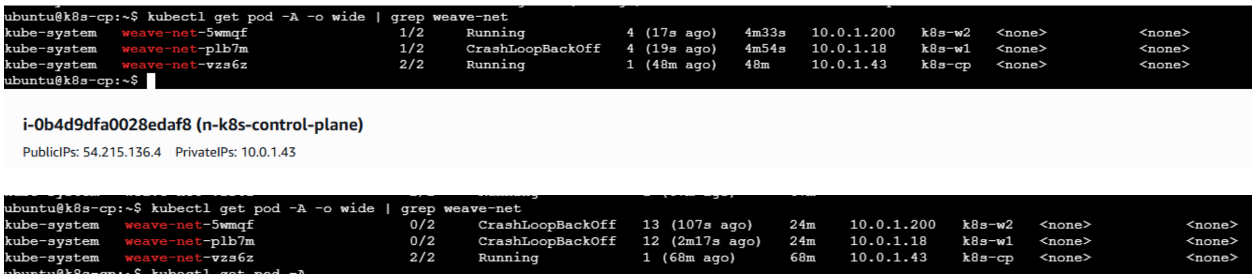
```
ubuntu@k8s-cp:~$ kubectl get pod -A -o wide
NAMESPACE     NAME                                READY   STATUS    RESTARTS       AGE     IP           NODE     NOMINATED NODE   READINESS GATES
kube-system   coredns-76f75df574-548cn            1/1     Running   0              92m     10.32.0.3    k8s-cp   <none>           <none>
kube-system   coredns-76f75df574-ktklx            1/1     Running   0              92m     10.32.0.2    k8s-cp   <none>           <none>
kube-system   etcd-k8s-cp                         1/1     Running   196            93m     10.0.1.43    k8s-cp   <none>           <none>
kube-system   kube-apiserver-k8s-cp               1/1     Running   197            93m     10.0.1.43    k8s-cp   <none>           <none>
kube-system   kube-controller-manager-k8s-cp      1/1     Running   0              93m     10.0.1.43    k8s-cp   <none>           <none>
kube-system   kube-proxy-hwcsr                    1/1     Running   2 (65s ago)    2m23s   10.0.1.18    k8s-w1   <none>           <none>
kube-system   kube-proxy-pxf94                    1/1     Running   0              92m     10.0.1.43    k8s-cp   <none>           <none>
kube-system   kube-proxy-ts7mw                    1/1     Running   2 (27s ago)    2m2s    10.0.1.200   k8s-w2   <none>           <none>
kube-system   kube-scheduler-k8s-cp               1/1     Running   201            93m     10.0.1.43    k8s-cp   <none>           <none>
kube-system   weave-net-5wmqf                     2/2     Running   0              2m2s    10.0.1.200   k8s-w2   <none>           <none>
kube-system   weave-net-plb7m                     2/2     Running   2 (11s ago)    2m23s   10.0.1.18    k8s-w1   <none>           <none>
kube-system   weave-net-vzs6z                     2/2     Running   1 (46m ago)    46m     10.0.1.43    k8s-cp   <none>           <none>
ubuntu@k8s-cp:~$
```

After both worker nodes joined the cluster, the cluster state was verified. This confirms:

- Pods distributed across nodes
- System components running on appropriate nodes

Note: At this stage, the cluster was functional but experienced networking issues.

5.11 Identifying Weave Net Pod Failures



```
ubuntu@k8s-cp:~$ kubectl get pod -A -o wide | grep weave-net
kube-system   weave-net-5wmqf                     1/2     Running            4 (17s ago)   4m33s   10.0.1.200   k8s-w2   <none>           <none>
kube-system   weave-net-plb7m                     1/2     CrashLoopBackOff   4 (19s ago)   4m54s   10.0.1.18    k8s-w1   <none>           <none>
kube-system   weave-net-vzs6z                     2/2     Running            1 (48m ago)   48m     10.0.1.43    k8s-cp   <none>           <none>
ubuntu@k8s-cp:~$
```

i-0b4d9dfa0028edaf8 (n-k8s-control-plane)

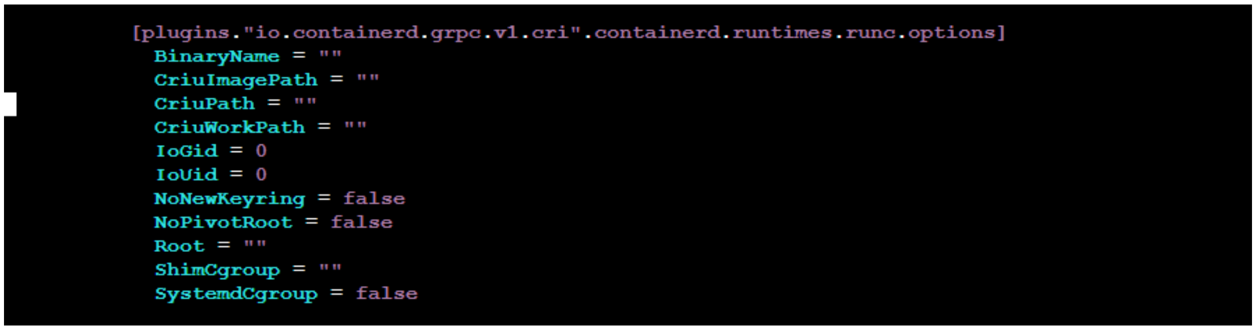
PublicIPs: 54.215.136.4   PrivateIPs: 10.0.1.43

```
ubuntu@k8s-cp:~$ kubectl get pod -A -o wide | grep weave-net
kube-system   weave-net-5wmqf                     0/2     CrashLoopBackOff   13 (107s ago)    24m   10.0.1.200   k8s-w2   <none>           <none>
kube-system   weave-net-plb7m                     0/2     CrashLoopBackOff   12 (2m17s ago)   24m   10.0.1.18    k8s-w1   <none>           <none>
kube-system   weave-net-vzs6z                     2/2     Running            1 (68m ago)      68m   10.0.1.43    k8s-cp   <none>           <none>
```

Weave Net pods were observed in a crash loop state. This indicated a networking or container runtime configuration issue affecting the CNI plugin.

Further investigation revealed a mismatch between kubelet and containerd cgroup settings.

5.12 Containerd Configuration Issue Identification



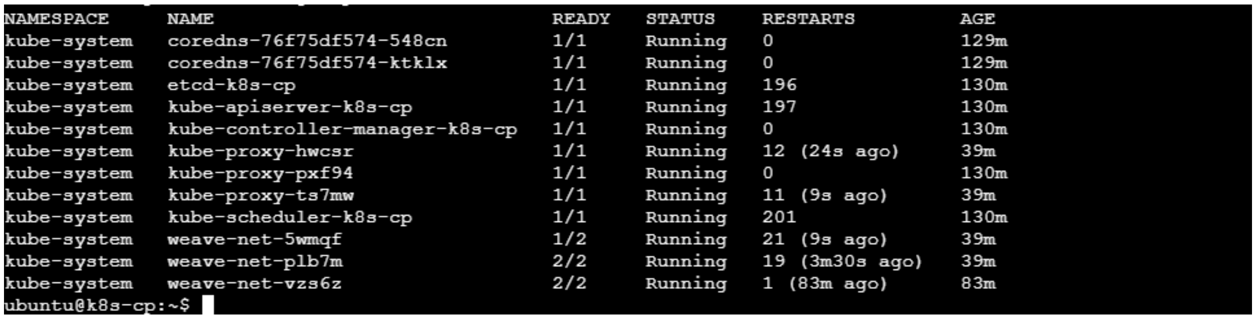
```
        [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
          BinaryName = ""
          CriuImagePath = ""
          CriuPath = ""
          CriuWorkPath = ""
          IoGid = 0
          IoUid = 0
          NoNewKeyring = false
          NoPivotRoot = false
          Root = ""
          ShimCgroup = ""
          SystemdCgroup = false
```

The containerd configuration file was inspected and showed:

- SystemdCgroup set to **false**

This setting is incompatible with kubelet's default systemd cgroup driver on Ubuntu 24.04.

5.13 Applying the Fix and Validating the Cluster



```
NAMESPACE     NAME                                READY   STATUS    RESTARTS          AGE
kube-system   coredns-76f75df574-548cn            1/1     Running   0                 129m
kube-system   coredns-76f75df574-ktklx            1/1     Running   0                 129m
kube-system   etcd-k8s-cp                         1/1     Running   196               130m
kube-system   kube-apiserver-k8s-cp               1/1     Running   197               130m
kube-system   kube-controller-manager-k8s-cp      1/1     Running   0                 130m
kube-system   kube-proxy-hwcsr                    1/1     Running   12 (24s ago)      39m
kube-system   kube-proxy-pxf94                    1/1     Running   0                 130m
kube-system   kube-proxy-ts7mw                    1/1     Running   11 (9s ago)       39m
kube-system   kube-scheduler-k8s-cp               1/1     Running   201               130m
kube-system   weave-net-5wmqf                     1/2     Running   21 (9s ago)       39m
kube-system   weave-net-plb7m                     2/2     Running   19 (3m30s ago)    39m
kube-system   weave-net-vzs6z                     2/2     Running   1 (83m ago)       83m
ubuntu@k8s-cp:~$
```

The containerd configuration was updated to:

- SystemdCgroup = **true**

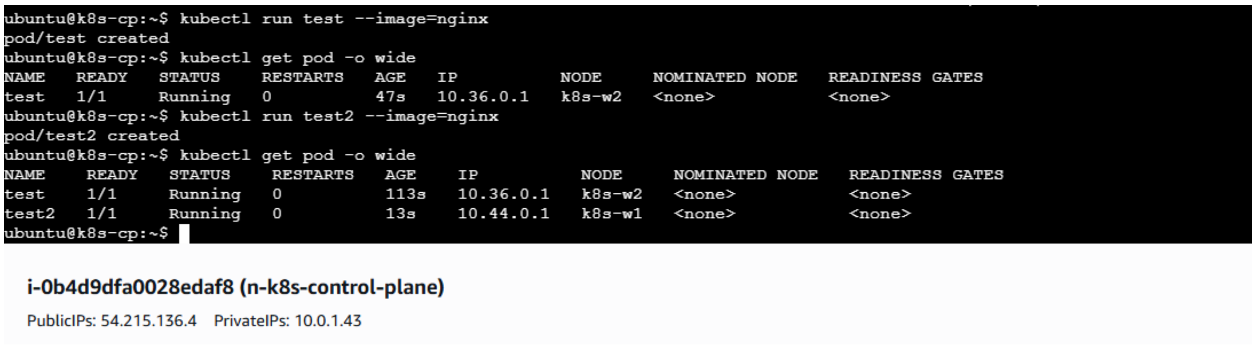
After restarting containerd and kubelet, all Weave Net pods transitioned to a **Running** and **Ready** state.

This final verification confirms:

- Stable pod networking
- Healthy Kubernetes cluster
- Successful resolution of runtime and CNI issues

6. Post-Cluster Validation

After completing the Kubernetes installation and resolving networking issues, basic validation tests were performed to confirm that the cluster was operational. These tests focused on pod scheduling, worker node availability, and cluster networking.



```
ubuntu@k8s-cp:~$ kubectl run test --image=nginx
pod/test created
ubuntu@k8s-cp:~$ kubectl get pod -o wide
NAME   READY   STATUS    RESTARTS   AGE   IP          NODE     NOMINATED NODE   READINESS GATES
test   1/1     Running   0          47s   10.36.0.1   k8s-w2   <none>           <none>
ubuntu@k8s-cp:~$ kubectl run test2 --image=nginx
pod/test2 created
ubuntu@k8s-cp:~$ kubectl get pod -o wide
NAME    READY   STATUS    RESTARTS   AGE    IP          NODE     NOMINATED NODE   READINESS GATES
test    1/1     Running   0          113s   10.36.0.1   k8s-w2   <none>           <none>
test2   1/1     Running   0          13s    10.44.0.1   k8s-w1   <none>           <none>
ubuntu@k8s-cp:~$
```

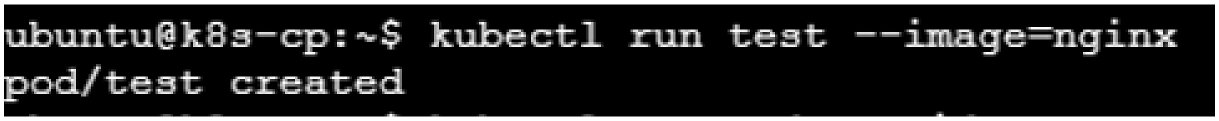
i-0b4d9dfa0028edaf8 (n-k8s-control-plane)

PublicIPs: 54.215.136.4   PrivateIPs: 10.0.1.43

6.1 Deploying a Test Pod

A simple test pod was created using the official nginx image to validate that the cluster could accept and schedule workloads.

Command executed:
kubectl run test --image=nginx



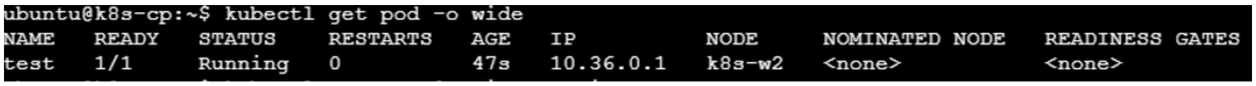
```
ubuntu@k8s-cp:~$ kubectl run test --image=nginx
pod/test created
```

Successful pod creation confirms that the control plane and scheduler are functioning correctly.

6.2 Verifying Pod Scheduling

The pod status was verified using ***kubectl get pods -o wide*** command. The output shows:

- Pod test in **Running** state
- Pod scheduled on worker node k8s-w2
- Pod assigned an IP address from the cluster network

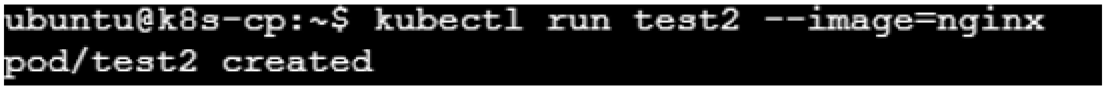


```
ubuntu@k8s-cp:~$ kubectl get pod -o wide
NAME   READY   STATUS    RESTARTS   AGE   IP          NODE     NOMINATED NODE   READINESS GATES
test   1/1     Running   0          47s   10.36.0.1   k8s-w2   <none>           <none>
```

This confirms that worker nodes are active and pod networking is working as expected.

6.3 Validating Workload Distribution

A second test pod was created to further validate scheduling behavior.



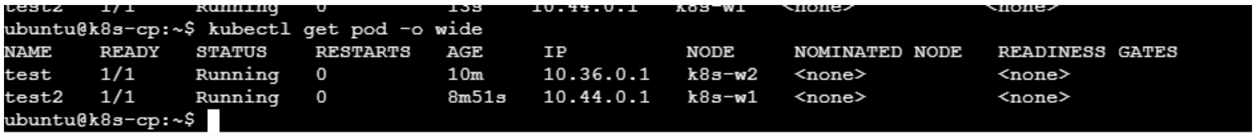
```
ubuntu@k8s-cp:~$ kubectl run test2 --image=nginx
pod/test2 created
```

Command executed:

- *kubectl run test2 --image=nginx*

The pod list was reviewed again using ***kubectl get pods -o wide*** command. The output confirms:

- test pod running on k8s-w2
- test2 pod running on k8s-w1
- Both pods in **Running** and **Ready** state
- Unique pod IPs assigned by the CNI



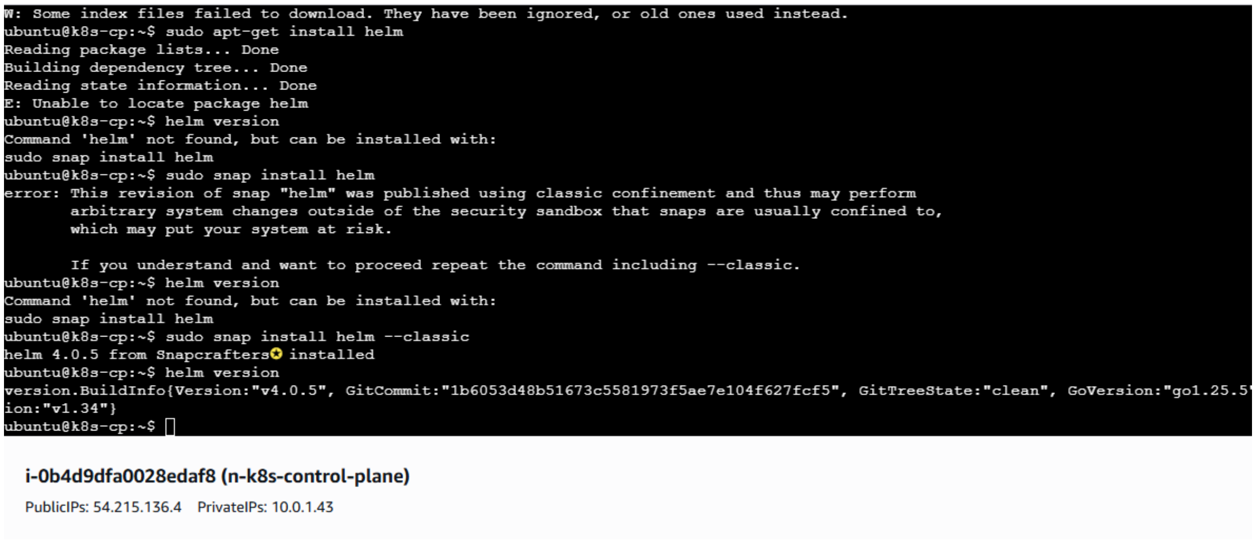
```
test2   1/1     Running   0          13s    10.44.0.1   k8s-w1   <none>           <none>
ubuntu@k8s-cp:~$ kubectl get pod -o wide
NAME    READY   STATUS    RESTARTS   AGE     IP          NODE     NOMINATED NODE   READINESS GATES
test    1/1     Running   0          10m     10.36.0.1   k8s-w2   <none>           <none>
test2   1/1     Running   0          8m51s   10.44.0.1   k8s-w1   <none>           <none>
ubuntu@k8s-cp:~$
```

This demonstrates proper scheduler operation and successful workload distribution across worker nodes.

7. Helm Installation and Verification

Helm is a Kubernetes package manager that simplifies deploying and managing applications via charts. This section documents installing Helm on the control plane node and verifying its functionality.

7.1 Helm Installation



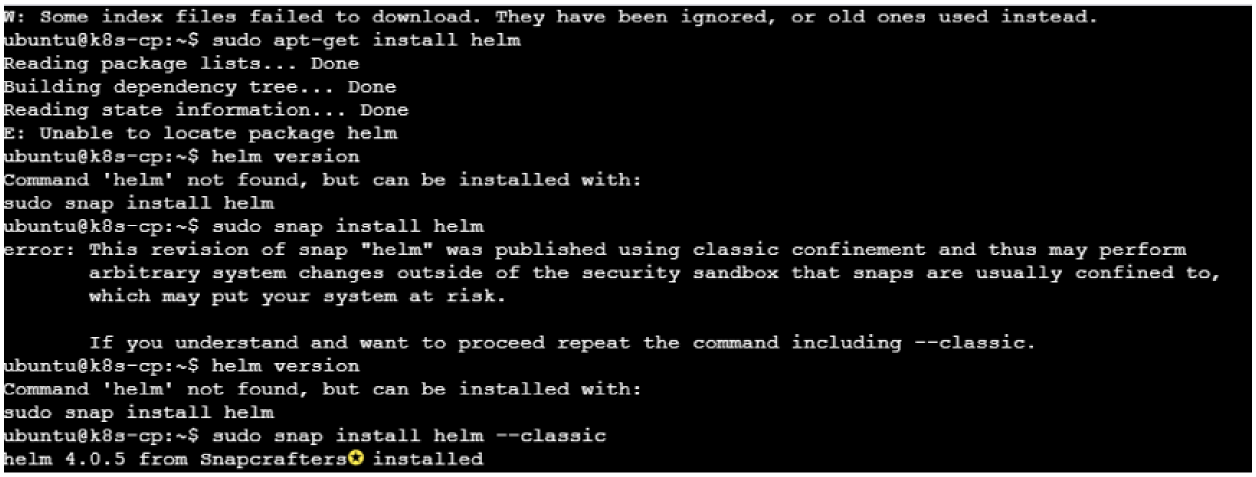
```
W: Some index files failed to download. They have been ignored, or old ones used instead.
ubuntu@k8s-cp:~$ sudo apt-get install helm
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
E: Unable to locate package helm
ubuntu@k8s-cp:~$ helm version
Command 'helm' not found, but can be installed with:
sudo snap install helm
ubuntu@k8s-cp:~$ sudo snap install helm
error: This revision of snap "helm" was published using classic confinement and thus may perform
       arbitrary system changes outside of the security sandbox that snaps are usually confined to,
       which may put your system at risk.

       If you understand and want to proceed repeat the command including --classic.
ubuntu@k8s-cp:~$ helm version
Command 'helm' not found, but can be installed with:
sudo snap install helm
ubuntu@k8s-cp:~$ sudo snap install helm --classic
helm 4.0.5 from Snapcrafters✪ installed
ubuntu@k8s-cp:~$ helm version
version.BuildInfo{Version:"v4.0.5", GitCommit:"1b6053d48b51673c5581973f5ae7e104f627fcf5", GitTreeState:"clean", GoVersion:"go1.25.5'
ion:"v1.34"}
ubuntu@k8s-cp:~$ 
```

i-0b4d9dfa0028edaf8 (n-k8s-control-plane)

PublicIPs: 54.215.136.4 PrivateIPs: 10.0.1.43

Helm was initially attempted to be installed via the default Ubuntu package manager.

- ***sudo apt-get install helm*** failed with an error. E: Unable to locate package helm because Helm is not available in the default Ubuntu repositories.
- ***sudo snap install helm –classic:*** Snap was then used to install Helm using the command

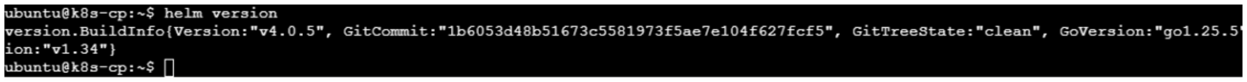


```
W: Some index files failed to download. They have been ignored, or old ones used instead.
ubuntu@k8s-cp:~$ sudo apt-get install helm
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
E: Unable to locate package helm
ubuntu@k8s-cp:~$ helm version
Command 'helm' not found, but can be installed with:
sudo snap install helm
ubuntu@k8s-cp:~$ sudo snap install helm
error: This revision of snap "helm" was published using classic confinement and thus may perform
       arbitrary system changes outside of the security sandbox that snaps are usually confined to,
       which may put your system at risk.

       If you understand and want to proceed repeat the command including --classic.
ubuntu@k8s-cp:~$ helm version
Command 'helm' not found, but can be installed with:
sudo snap install helm
ubuntu@k8s-cp:~$ sudo snap install helm --classic
helm 4.0.5 from Snapcrafters✪ installed
```

This ensures Helm is installed correctly and ready for Kubernetes usage.

7.2 Helm Verification

The installation was verified by running ***helm version***



```
ubuntu@k8s-cp:~$ helm version
version.BuildInfo{Version:"v4.0.5", GitCommit:"1b6053d48b51673c5581973f5ae7e104f627fcf5", GitTreeState:"clean", GoVersion:"go1.25.5
ion:"v1.34"}
ubuntu@k8s-cp:~$ 
```

The output confirms:

- Helm is installed and operational
- Version installed: **v4.0.5**
- Build information and Go version are displayed

This final verification ensures Helm can now be used to manage charts and deploy applications on the cluster.

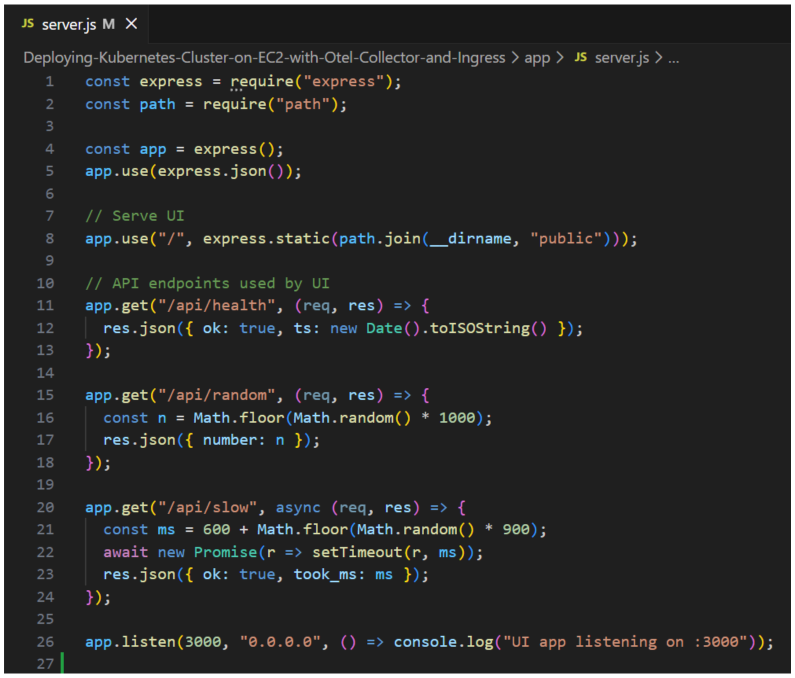
8. Application Code Overview

The deployed application is a Node.js-based UI service that demonstrates Kubernetes workload management and observability with OpenTelemetry. It consists of a backend API, a frontend UI, and integrated tracing.

8.1 Backend: server.js and tracing.js

The server.js file implements an **Express** web server that serves both the UI and API endpoints.

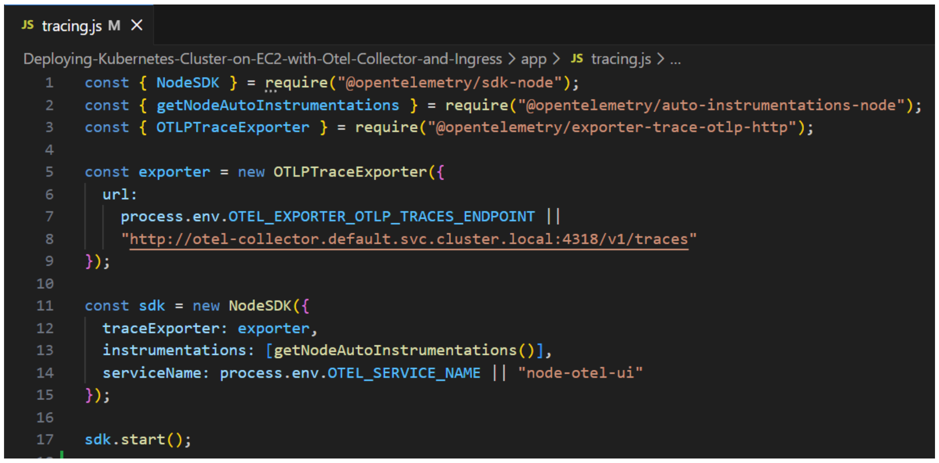
- **UI serving:** Static files from the public directory
- **API endpoints:**
  - ***/api/health*** – returns service health and timestamp
  - ***/api/random*** – returns a random number
  - ***/api/slow*** – simulates a slow request with a random delay
- **Port:** Listens on 0.0.0.0:3000 to allow Kubernetes networking access


server.js M
Deploying-Kubernetes-Cluster-on-EC2-with-Otel-Collector-and-Ingress > app > server.js > ...

```
const express = require("express");
const path = require("path");

const app = express();
app.use(express.json());

// Serve UI
app.use("/", express.static(path.join(__dirname, "public")));

// API endpoints used by UI
app.get("/api/health", (req, res) => {
  res.json({ ok: true, ts: new Date().toISOString() });
});

app.get("/api/random", (req, res) => {
  const n = Math.floor(Math.random() * 1000);
  res.json({ number: n });
});

app.get("/api/slow", async (req, res) => {
  const ms = 600 + Math.floor(Math.random() * 900);
  await new Promise(r => setTimeout(r, ms));
  res.json({ ok: true, took_ms: ms });
});

app.listen(3000, "0.0.0.0", () => console.log("UI app listening on :3000"));
```

The **tracing.js** integrates **OpenTelemetry** for observability.

- Configures an OTLP trace exporter pointing to in-cluster OpenTelemetry Collector
- Automatically instruments Node.js modules
- Enables tracing of API requests and performance for monitoring


tracing.js M ×

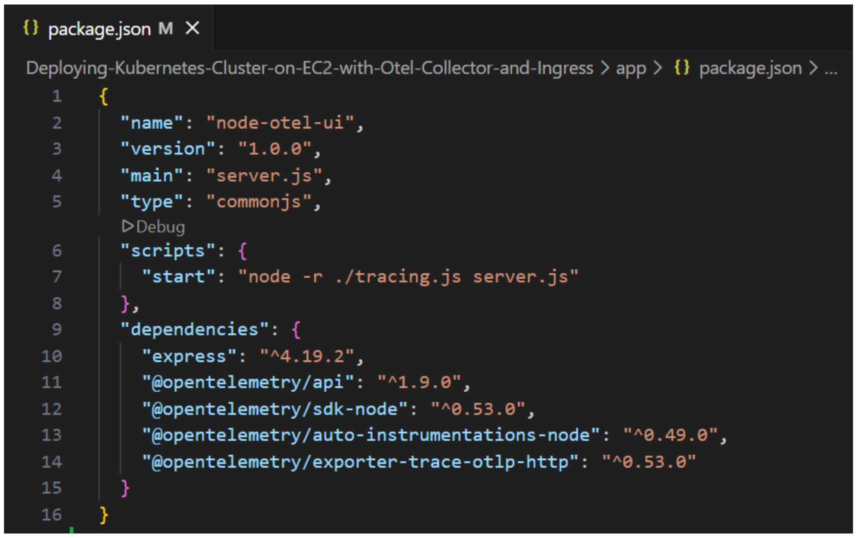
Deploying-Kubernetes-Cluster-on-EC2-with-Otel-Collector-and-Ingress > app > tracing.js > ...

```javascript
const { NodeSDK } = require("@opentelemetry/sdk-node");
const { getNodeAutoInstrumentations } = require("@opentelemetry/auto-instrumentations-node");
const { OTLPTraceExporter } = require("@opentelemetry/exporter-trace-otlp-http");

const exporter = new OTLPTraceExporter({
  url:
    process.env.OTEL_EXPORTER_OTLP_TRACES_ENDPOINT ||
    "http://otel-collector.default.svc.cluster.local:4318/v1/traces"
});

const sdk = new NodeSDK({
  traceExporter: exporter,
  instrumentations: [getNodeAutoInstrumentations()],
  serviceName: process.env.OTEL_SERVICE_NAME || "node-otel-ui"
});

sdk.start();
```

8.2 Package Management: package.json

package.json defines dependencies and scripts.

- **Core dependencies:** express for server, OpenTelemetry packages for tracing
- **Start script:** Runs server.js with tracing.js preloaded
- Package lock ensures consistent builds across environments


package.json M ×

Deploying-Kubernetes-Cluster-on-EC2-with-Otel-Collector-and-Ingress > app > package.json > ...

```json
{
  "name": "node-otel-ui",
  "version": "1.0.0",
  "main": "server.js",
  "type": "commonjs",
  "scripts": {
    "start": "node -r ./tracing.js server.js"
  },
  "dependencies": {
    "express": "^4.19.2",
    "@opentelemetry/api": "^1.9.0",
    "@opentelemetry/sdk-node": "^0.53.0",
    "@opentelemetry/auto-instrumentations-node": "^0.49.0",
    "@opentelemetry/exporter-trace-otlp-http": "^0.53.0"
  }
}
```

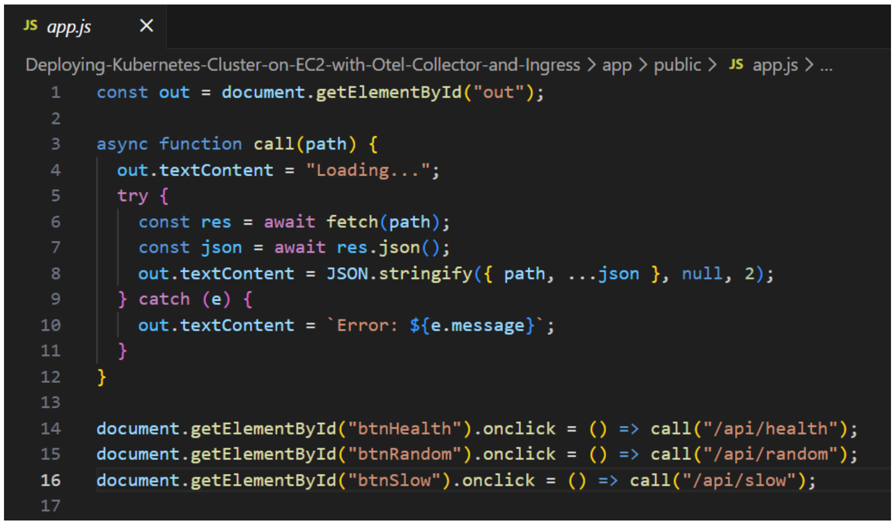
This configuration supports both local testing and containerized deployment.

8.3 Frontend: public Directory

The frontend provides a simple UI for interacting with the API:

- index.html: buttons for health, random number, and slow request endpoints

- app.js: handles button clicks and displays API responses dynamically
- styles.css: basic UI styling for readability and clarity


app.js

Deploying-Kubernetes-Cluster-on-EC2-with-Otel-Collector-and-Ingress > app > public > app.js > ...

```javascript
const out = document.getElementById("out");

async function call(path) {
  out.textContent = "Loading...";
  try {
    const res = await fetch(path);
    const json = await res.json();
    out.textContent = JSON.stringify({ path, ...json }, null, 2);
  } catch (e) {
    out.textContent = `Error: ${e.message}`;
  }
}

document.getElementById("btnHealth").onclick = () => call("/api/health");
document.getElementById("btnRandom").onclick = () => call("/api/random");
document.getElementById("btnSlow").onclick = () => call("/api/slow");
```

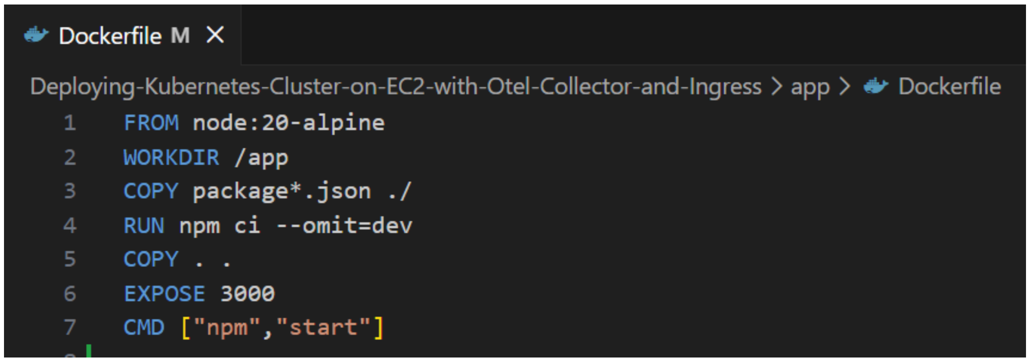
The UI demonstrates request handling and allows visual verification of OpenTelemetry traces.

8.4 Containerization: Dockerfile and Image Distribution

The application is containerized using Docker to ensure consistency across development and Kubernetes deployment environments.

8.4.1 Dockerfile Overview:

- **Base image:** node:20-alpine
- **Dependency installation:** Uses npm ci --omit=dev to install dependencies
- **Application port:** Exposes port 3000, later mapped by a Kubernetes Service.
- **Entrypoint:** *npm start* launches the Node.js server with OpenTelemetry auto-instrumentation enabled via tracing.js.


Dockerfile M

Deploying-Kubernetes-Cluster-on-EC2-with-Otel-Collector-and-Ingress > app > Dockerfile

```dockerfile
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev
COPY . .
EXPOSE 3000
CMD ["npm","start"]
```

This container image is referenced by the Kubernetes Deployment and integrates seamlessly with the cluster's networking and observability configuration.

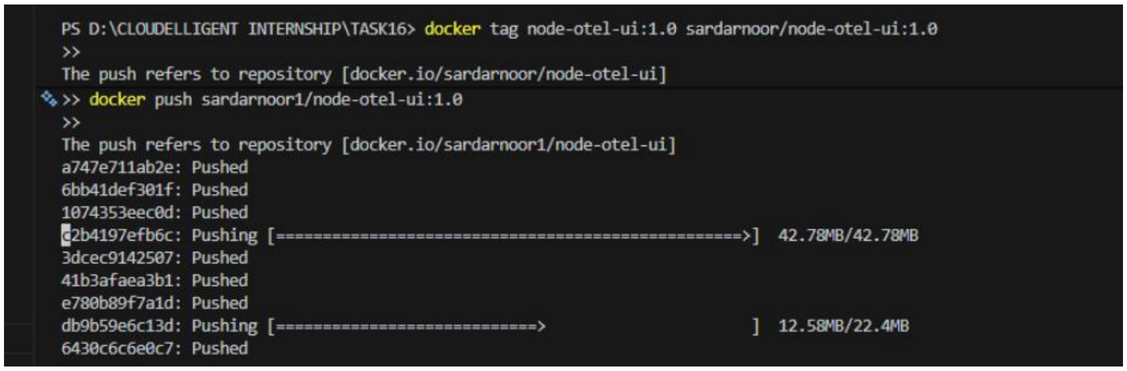
8.4.2 Docker Hub Image Publishing:

Since this Kubernetes cluster is self-managed (kubeadm on EC2) and does not automatically authenticate to Amazon ECR, the container image was pushed to **Docker Hub** as a public repository to simplify image pulls from worker nodes.

Commands executed (local machine):
docker tag node-otel-ui:1.0 sardarnoor1/node-otel-ui:1.0
docker push sardarnoor1/node-otel-ui:1.0

Outcome:

- The image was successfully pushed to Docker Hub.
- Worker nodes were able to pull the image without additional registry credentials.
- This resolved ImagePullBackOff issues encountered with private registries.



```
PS D:\CLOUDELLIGENT INTERNSHIP\TASK16> docker tag node-otel-ui:1.0 sardarnoor/node-otel-ui:1.0
>>
The push refers to repository [docker.io/sardarnoor/node-otel-ui]
>> docker push sardarnoor1/node-otel-ui:1.0
>>
The push refers to repository [docker.io/sardarnoor1/node-otel-ui]
a747e711ab2e: Pushed
6bb41def301f: Pushed
1074353eec0d: Pushed
c2b4197efb6c: Pushing [==================================================>]  42.78MB/42.78MB
3dcec9142507: Pushed
41b3afaea3b1: Pushed
e780b89f7a1d: Pushed
db9b59e6c13d: Pushing [============================>                      ]  12.58MB/22.4MB
6430c6c6e0c7: Pushed
```

This container image is deployed as a **Kubernetes Deployment** and is compatible with the cluster's networking and observability setup.

8.5 Summary

The application demonstrates:

- Full-stack Node.js functionality with API and UI
- Integration with OpenTelemetry for tracing and monitoring
- Containerization for Kubernetes deployment
- Clear separation of backend, frontend, and observability concerns

This structure allows easy scaling, monitoring, and extension of the application in a Kubernetes environment.

9. Kubernetes Manifest YAML Files

The cluster deploys three key sets of Kubernetes resources: the Node.js application, the Ingress routing, and the OpenTelemetry Collector. Each is defined in its own YAML manifest for clarity and modularity.

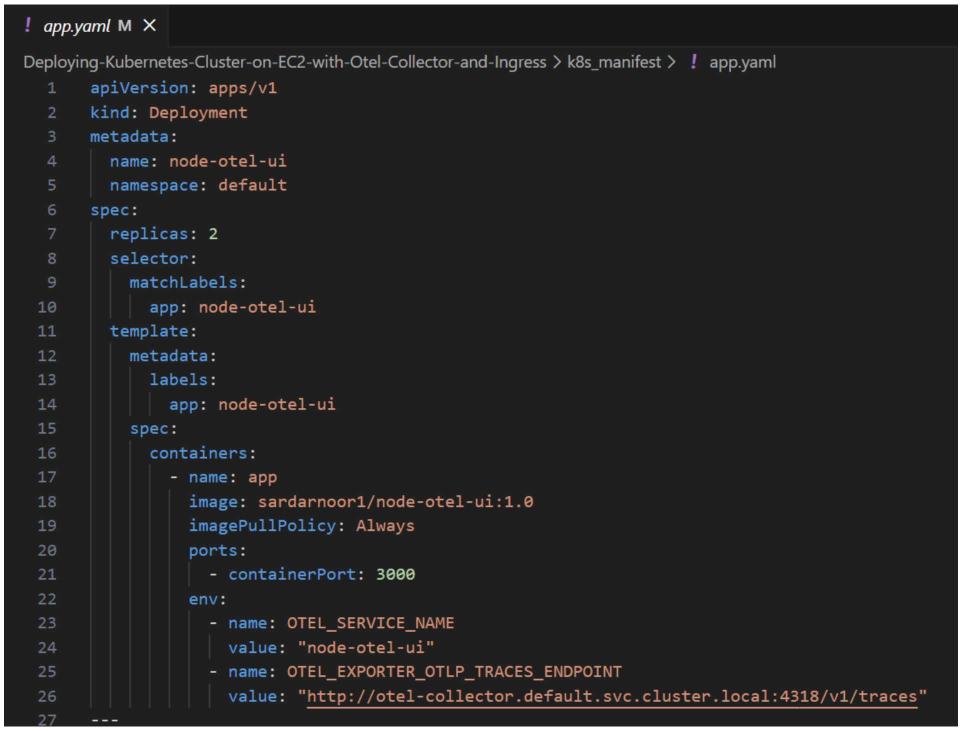
9.1 Application Deployment and Service (app.yaml)

The app.yaml manifest defines **Deployment** and **Service** resource.

9.1.1 Deployment

- **Replicas: 2**

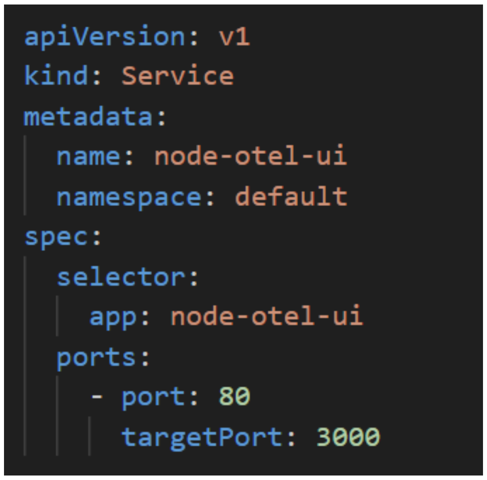
  Ensures two pods run for high availability
- **Labels: app**
  - **node-otel-ui** for service selection and scheduling
- **Container**
  - **sardarnoor1/node-otel-ui:1.0** running on port 3000
- **Environment variables:**
  - **OTEL_SERVICE_NAME**: sets the service name for tracing
  - **OTEL_EXPORTER_OTLP_TRACES_ENDPOINT**: points to the in-cluster OpenTelemetry Collector


! app.yaml M ×
Deploying-Kubernetes-Cluster-on-EC2-with-Otel-Collector-and-Ingress > k8s_manifest > ! app.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-otel-ui
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: node-otel-ui
  template:
    metadata:
      labels:
        app: node-otel-ui
    spec:
      containers:
        - name: app
          image: sardarnoor1/node-otel-ui:1.0
          imagePullPolicy: Always
          ports:
            - containerPort: 3000
          env:
            - name: OTEL_SERVICE_NAME
              value: "node-otel-ui"
            - name: OTEL_EXPORTER_OTLP_TRACES_ENDPOINT
              value: "http://otel-collector.default.svc.cluster.local:4318/v1/traces"
---
```

9.1.2 Service

- type: ClusterIP (default) exposes the deployment internally
- Maps **port 80** externally to **container port 3000** internally
- Selects pods based on app: node-otel-ui label

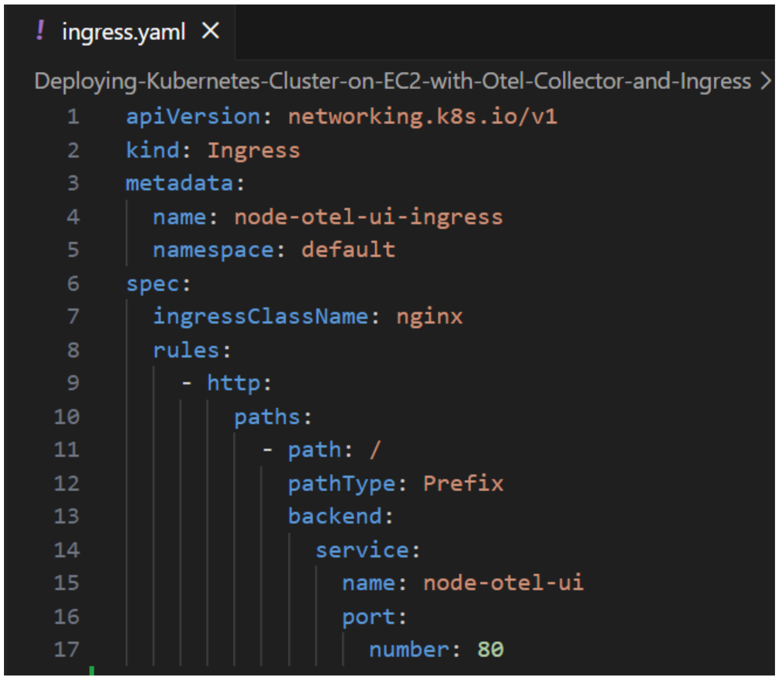


```yaml
apiVersion: v1
kind: Service
metadata:
  name: node-otel-ui
  namespace: default
spec:
  selector:
    app: node-otel-ui
  ports:
    - port: 80
      targetPort: 3000
```

This setup allows the application to run reliably on multiple pods and be discoverable by other cluster services.

9.2 Ingress Routing (ingress.yaml)

The ingress.yaml manifest defines an **Ingress** resource.



ingress.yaml

Deploying-Kubernetes-Cluster-on-EC2-with-Otel-Collector-and-Ingress >

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: node-otel-ui-ingress
  namespace: default
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: node-otel-ui
                port:
                  number: 80
```

- Routes HTTP traffic from / to the node-otel-ui service
- Uses ingressClassName: nginx to specify the NGINX Ingress Controller
- Supports external access without exposing NodePorts directly

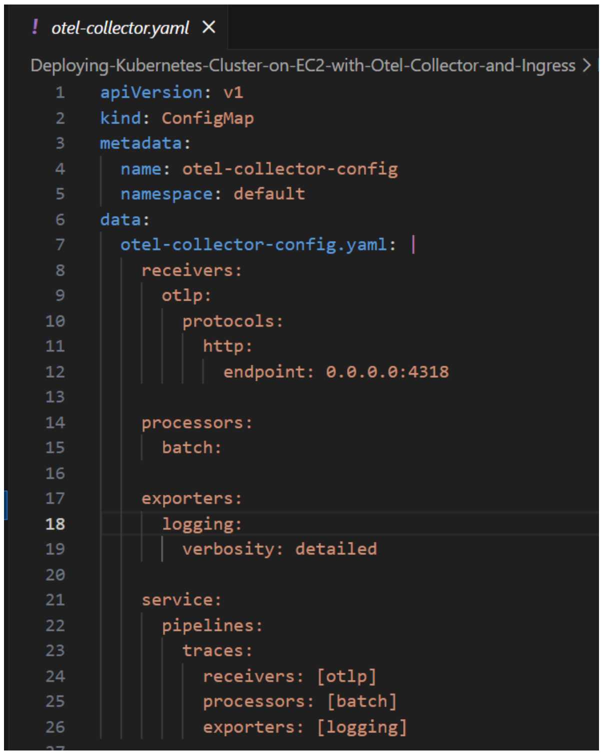
This allows the application to be accessed via a single URL while enabling future path-based routing if needed.

9.3 OpenTelemetry Collector (otel-collector.yaml)

The otel-collector.yaml manifest deploys the OpenTelemetry Collector for observability.

9.3.1 ConfigMap

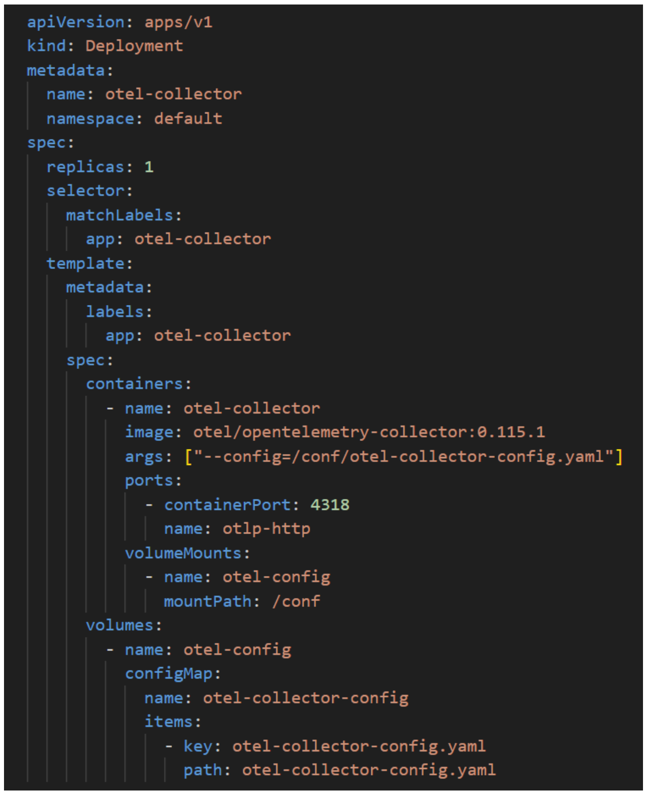
- Defines the collector configuration
- Receives OTLP traces from instrumented applications
- Uses a batch processor and logs traces with verbosity


otel-collector.yaml
Deploying-Kubernetes-Cluster-on-EC2-with-Otel-Collector-and-Ingress >

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: otel-collector-config
  namespace: default
data:
  otel-collector-config.yaml: |
    receivers:
      otlp:
        protocols:
          http:
            endpoint: 0.0.0.0:4318

    processors:
      batch:

    exporters:
      logging:
        verbosity: detailed

    service:
      pipelines:
        traces:
          receivers: [otlp]
          processors: [batch]
          exporters: [logging]
```

9.3.2 Deployment

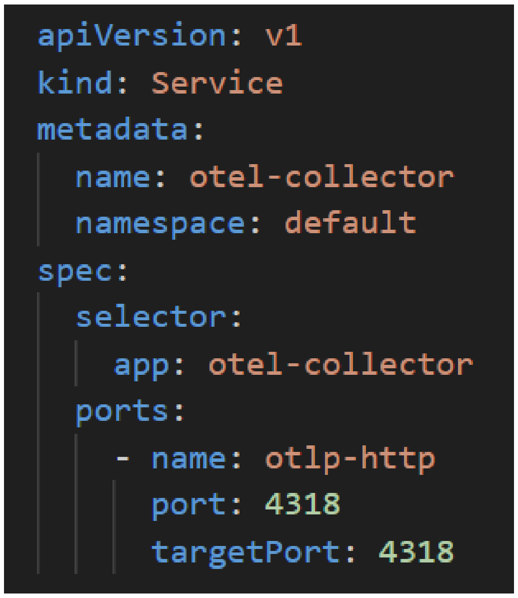
- Runs a single replica of the otel/opentelemetry-collector:0.115.1 container
- Mounts the collector configuration from the ConfigMap
- Exposes port 4318 for OTLP HTTP traffic



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: otel-collector
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: otel-collector
  template:
    metadata:
      labels:
        app: otel-collector
    spec:
      containers:
        - name: otel-collector
          image: otel/opentelemetry-collector:0.115.1
          args: ["--config=/conf/otel-collector-config.yaml"]
          ports:
            - containerPort: 4318
              name: otlp-http
          volumeMounts:
            - name: otel-config
              mountPath: /conf
      volumes:
        - name: otel-config
          configMap:
            name: otel-collector-config
            items:
              - key: otel-collector-config.yaml
                path: otel-collector-config.yaml
```

9.3.2 Service

- Exposes the collector internally on port 4318
- Allows pods to send traces to otel-collector.default.svc.cluster.local



```yaml
apiVersion: v1
kind: Service
metadata:
  name: otel-collector
  namespace: default
spec:
  selector:
    app: otel-collector
  ports:
    - name: otlp-http
      port: 4318
      targetPort: 4318
```

This setup ensures all application traces are collected and logged, enabling observability across the cluster.

10. Deploy and Verify OpenTelemetry Collector

10.1 Apply Collector Manifests

The OpenTelemetry Collector receives traces from the Node.js application and logs them for observability. To deploy it in the cluster:

Command executed:
kubectl apply -f otel-collector.yaml

Explanation:
This creates three Kubernetes objects.

1. **ConfigMap**: Defines the collector configuration (receivers, processors, exporters).
2. **Deployment**: Runs the collector pod(s).
3. **Service**: Exposes the collector internally on port 4318 for OTLP HTTP.

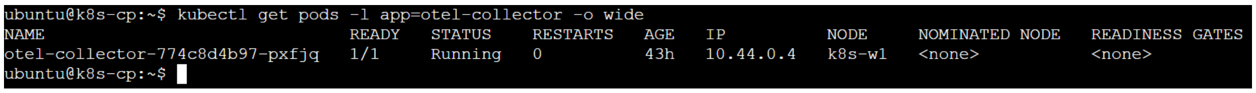
10.2 Verify Collector Pod Status

After applying the manifest, check that the collector pod is running:

Command executed:
kubectl get pods -l app=otel-collector -o wide

Explanation:

- Ensures the pod is **Running** and ready.
- Confirms that Kubernetes has scheduled the pod onto a node.
- Important columns to verify: READY = 1/1, STATUS = Running, NODE assigned.



```
ubuntu@k8s-cp:~$ kubectl get pods -l app=otel-collector -o wide
NAME                              READY   STATUS    RESTARTS   AGE   IP          NODE     NOMINATED NODE   READINESS GATES
otel-collector-774c8d4b97-pxfjq   1/1     Running   0          43h   10.44.0.4   k8s-w1   <none>           <none>
ubuntu@k8s-cp:~$ 
```

10.3 Verify Collector Service

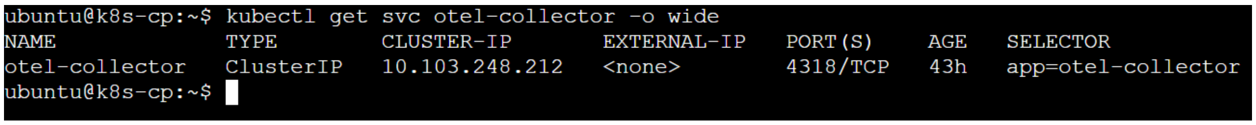
Check the internal service to ensure pods can be reached via ClusterIP.

Command executed:
kubectl get svc otel-collector -o wide

Explanation:

- Confirms the service exists and maps port 4318 to the collector pods.

- Used by the Node.js app to send OTLP traces.
- CLUSTER-IP shows the internal address.EXTERNAL-IP is <none>.



```
ubuntu@k8s-cp:~$ kubectl get svc otel-collector -o wide
NAME             TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE   SELECTOR
otel-collector   ClusterIP   10.103.248.212   <none>        4318/TCP   43h   app=otel-collector
ubuntu@k8s-cp:~$ 
```

11. Node.js UI Application Deployment

The objective is to deploy the application, verify pods are running, service is available, and OTEL instrumentation is configured.

11.1 Apply Application Manifest

Command executed:
kubectl apply -f app.yaml

Explanation:

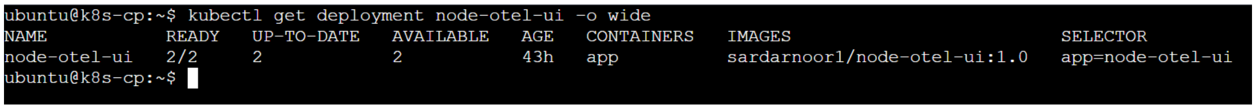
- Deploys the Node.js UI app using the Kubernetes manifest.
- Creates:
  - **Deployment** node-otel-ui (2 replicas)
  - **Service** node-otel-ui (ClusterIP, port 80 → container port 3000)
- Application is instrumented with OpenTelemetry to send traces to the collector.

11.2 Verify Deployment Status

Command executed:
kubectl get deployment node-otel-ui -o wide

Explanation:

- Confirms that the deployment is **READY 2/2**, meaning all replicas are up-to-date and available.
- Ensures Kubernetes has scheduled the pods correctly and the image is pulled successfully.



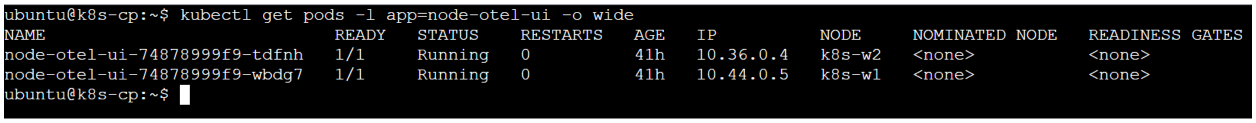
```
ubuntu@k8s-cp:~$ kubectl get deployment node-otel-ui -o wide
NAME           READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES                         SELECTOR
node-otel-ui   2/2     2            2           43h   app          sardarnoor1/node-otel-ui:1.0   app=node-otel-ui
ubuntu@k8s-cp:~$
```

11.3 Verify Pod Status

Command executed:
kubectl get pods -l app=node-otel-ui -o wide

Explanation:

- Confirms the individual pods are **Running** and ready.
- Helps identify if there are scheduling or image pull issues.



```
ubuntu@k8s-cp:~$ kubectl get pods -l app=node-otel-ui -o wide
NAME                          READY   STATUS    RESTARTS   AGE   IP          NODE     NOMINATED NODE   READINESS GATES
node-otel-ui-74878999f9-tdfnh   1/1     Running   0          41h   10.36.0.4   k8s-w2   <none>           <none>
node-otel-ui-74878999f9-wbdg7   1/1     Running   0          41h   10.44.0.5   k8s-w1   <none>           <none>
ubuntu@k8s-cp:~$
```

11.4 Verify Service

Command executed:

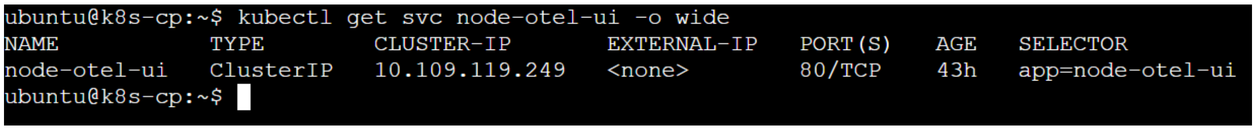
kubectl get svc node-otel-ui -o wide

Explanation:

Confirms that a **ClusterIP service** is created to allow internal cluster access to the application on port 80.

Key columns include:

- **CLUSTER-IP:** Internal IP for communication inside the cluster
- **PORT(S):** Service port mapping
- **SELECTOR:** Ensures traffic is sent to pods labeled app=node-otel-ui



```
ubuntu@k8s-cp:~$ kubectl get svc node-otel-ui -o wide
NAME           TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE   SELECTOR
node-otel-ui   ClusterIP   10.109.119.249   <none>        80/TCP    43h   app=node-otel-ui
ubuntu@k8s-cp:~$
```

12. Ingress Controller and Node.js UI Access

The objective is to expose the Node.js application externally using NGINX ingress.

12.1 Verify Ingress Controller Pods

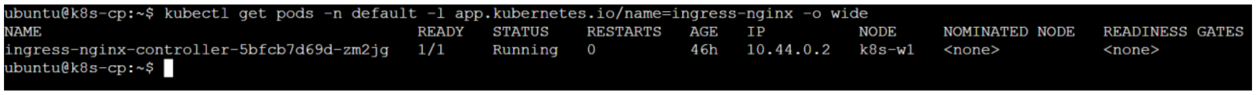
Command executed:

kubectl get pods -n default -l app.kubernetes.io/name=ingress-nginx -o wide

Explanation:

Confirms the **Ingress controller pod** is running and ready. Key columns include:

- **READY = 1/1** verifies that container running inside pod
- **STATUS = Running** confirms that pod started successfully
- **NODE** shows which EC2 instance is running the controller



```
ubuntu@k8s-cp:~$ kubectl get pods -n default -l app.kubernetes.io/name=ingress-nginx -o wide
NAME                                       READY   STATUS    RESTARTS   AGE   IP          NODE     NOMINATED NODE   READINESS GATES
ingress-nginx-controller-5bfcb7d69d-zm2jg   1/1     Running   0          46h   10.44.0.2   k8s-w1   <none>           <none>
ubuntu@k8s-cp:~$
```

12.2 Verify Ingress Controller Service (NodePort)

Command executed:

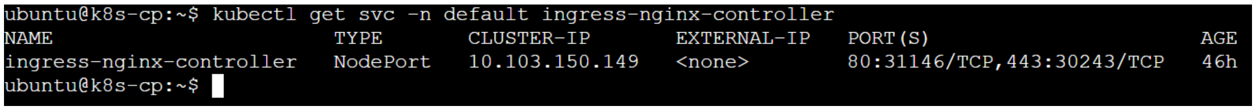
kubectl get svc -n default ingress-nginx-controller

Explanation:

Confirms the **Ingress controller service** exists and exposes HTTP/HTTPS via NodePort:

- HTTP NodePort: 31146
- HTTPS NodePort: 30243 (not used for this demo)

On kubeadm EC2 clusters, LoadBalancer services do **not automatically provision an AWS ELB**, so NodePort is used instead.



```
ubuntu@k8s-cp:~$ kubectl get svc -n default ingress-nginx-controller
NAME                       TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
ingress-nginx-controller   NodePort   10.103.150.149   <none>        80:31146/TCP,443:30243/TCP   46h
ubuntu@k8s-cp:~$
```

12.3 Apply Ingress Resource

Command executed (already applied):

kubectl apply -f ingress.yaml

Explanation:

- Routes external HTTP requests to the Node.js UI service (node-otel-ui:80).
- Ingress resource maps:
    - Host: *
    - Path: / → node-otel-ui:80
- Allows external access without exposing a NodePort directly on each pod.

12.4 Verify Ingress Resource

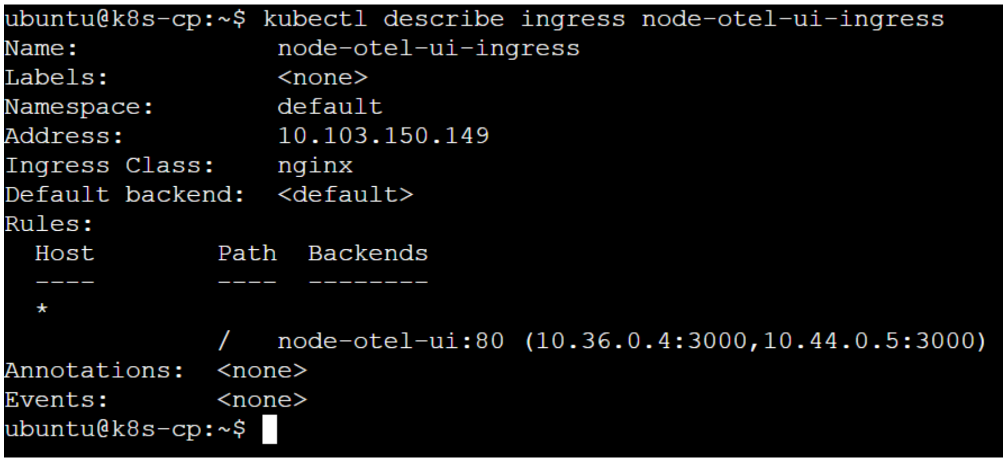
Command executed:

kubectl describe ingress node-otel-ui-ingress

Explanation:

Confirms that the Ingress resource is routing correctly to the backend service. Key information includes:

- **Rules:** / mapped to node-otel-ui:80
- **Backend endpoints:** IPs of the two app pods (10.36.0.4:3000, 10.44.0.5:3000)
- **Ingress Class:** nginx



```
ubuntu@k8s-cp:~$ kubectl describe ingress node-otel-ui-ingress
Name:             node-otel-ui-ingress
Labels:           <none>
Namespace:        default
Address:          10.103.150.149
Ingress Class:    nginx
Default backend:  <default>
Rules:
  Host        Path  Backends
  ----        ----  --------
  *
              /   node-otel-ui:80 (10.36.0.4:3000,10.44.0.5:3000)
Annotations:  <none>
Events:       <none>
ubuntu@k8s-cp:~$
```

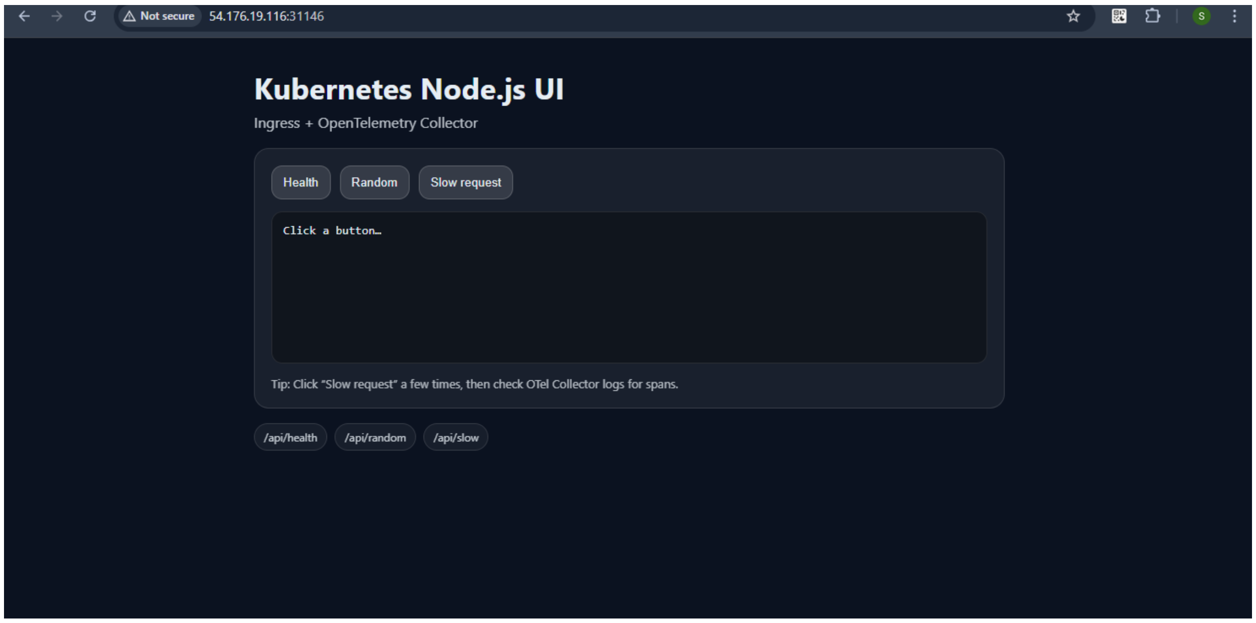
12.5 Access Node.js UI via NodePort

Steps:

- Open browser to the URL using **public IP of a node + NodePort**: http://<node-public-ip>:31146/
- For example: http://54.193.139.14:31146/

Explanation:

- Confirms the UI is accessible externally through the Ingress controller.
- Buttons (Health, Random, Slow request) can be clicked to generate requests and traces.



Kubernetes Node.js UI

Ingress + OpenTelemetry Collector

```
Click a button…
```

Tip: Click "Slow request" a few times, then check OTel Collector logs for spans.

13. OpenTelemetry Verification (End-to-End)

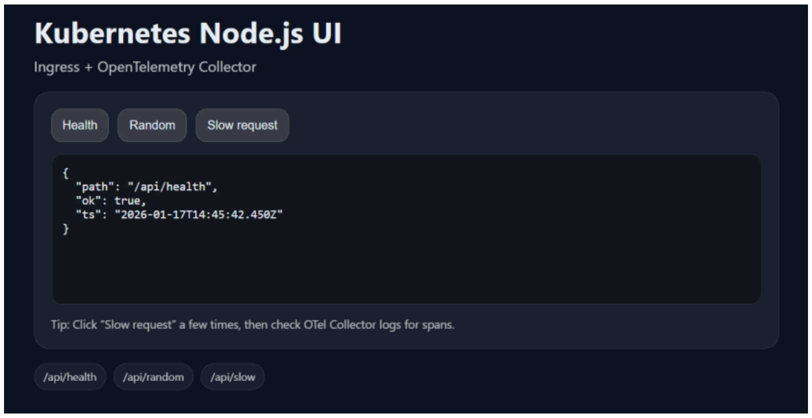
This section confirms that the **Node.js application** to **OTLP** to **OpenTelemetry Collector** pipeline is working and traces are successfully collected and exported.

13.1 Generate Traces from the UI

Steps to perform:

- Open the Node.js UI in your browser via the NodePort URL (http://<ec2-IP>:31146/).
- Click each button once: Health, Random, Slow request.
- This generates HTTP requests that are instrumented by OpenTelemetry.

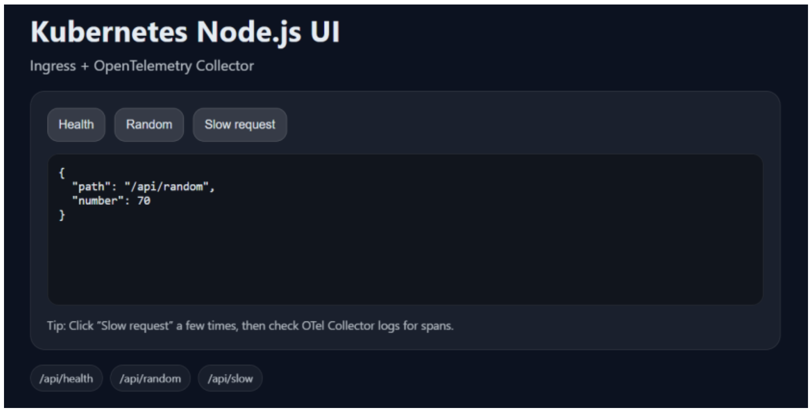
/Health: Tests /api/health endpoint



```
{
  "path": "/api/health",
  "ok": true,
  "ts": "2026-01-17T14:45:42.450Z"
}
```

Tip: Click "Slow request" a few times, then check OTel Collector logs for spans.

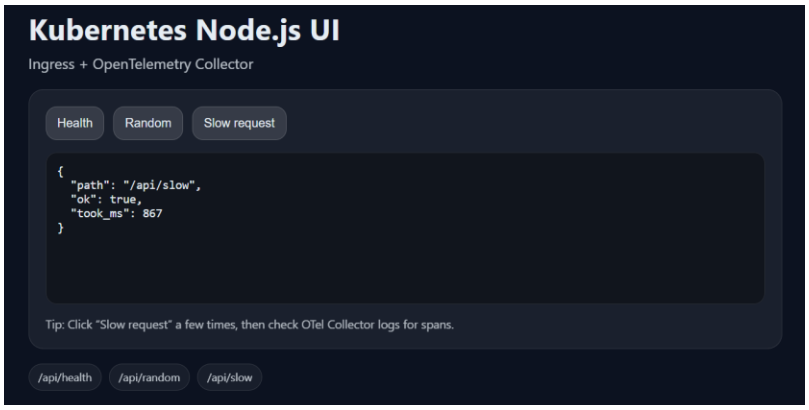
/Random: Tests /api/random endpoint



```
{
  "path": "/api/random",
  "number": 70
}
```

Tip: Click "Slow request" a few times, then check OTel Collector logs for spans.

/Slow (Slow request): Tests /api/slow endpoint (produces longer spans)


Kubernetes Node.js UI
Ingress + OpenTelemetry Collector
Health
Random
Slow request
Tip: Click "Slow request" a few times, then check OTel Collector logs for spans.
/api/health
/api/random
/api/slow

```
{
  "path": "/api/slow",
  "ok": true,
  "took_ms": 867
}
```

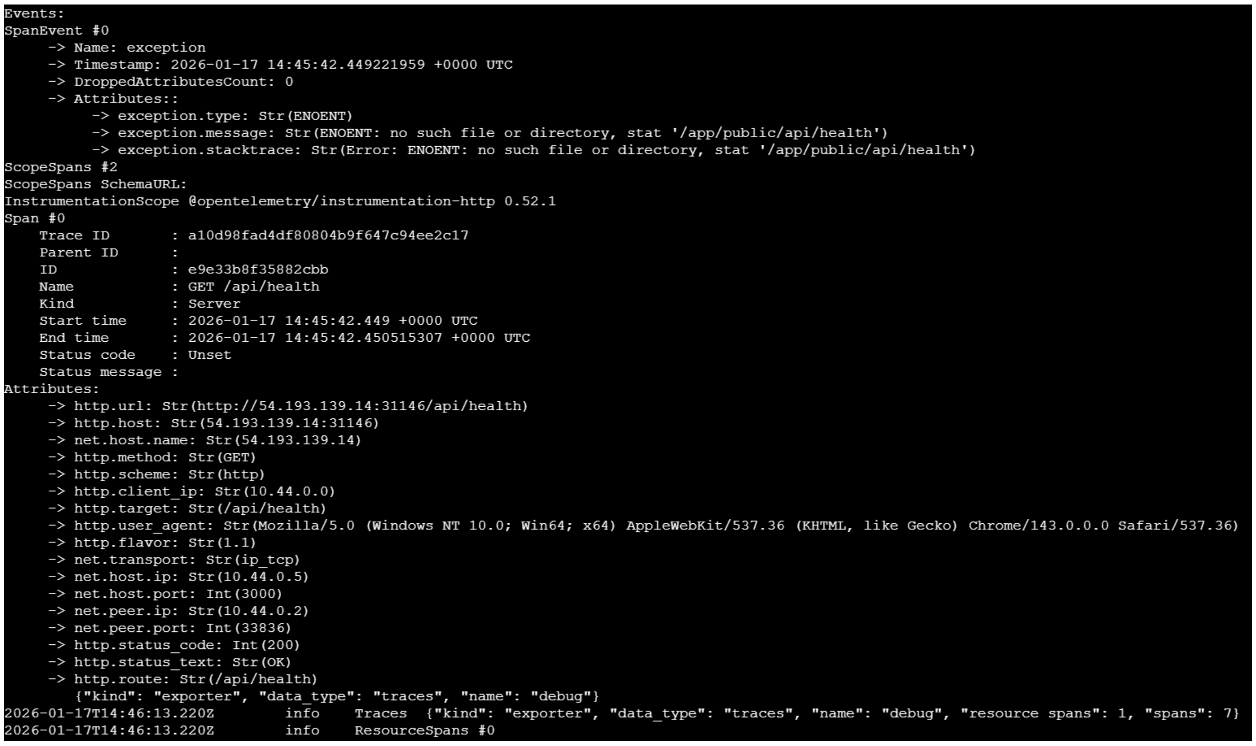
Each click generates one or more instrumented **spans** which are sent by **Node.js OTLP exporter** to the collector for processing.

13.2 Verify Collector Logs

Command:

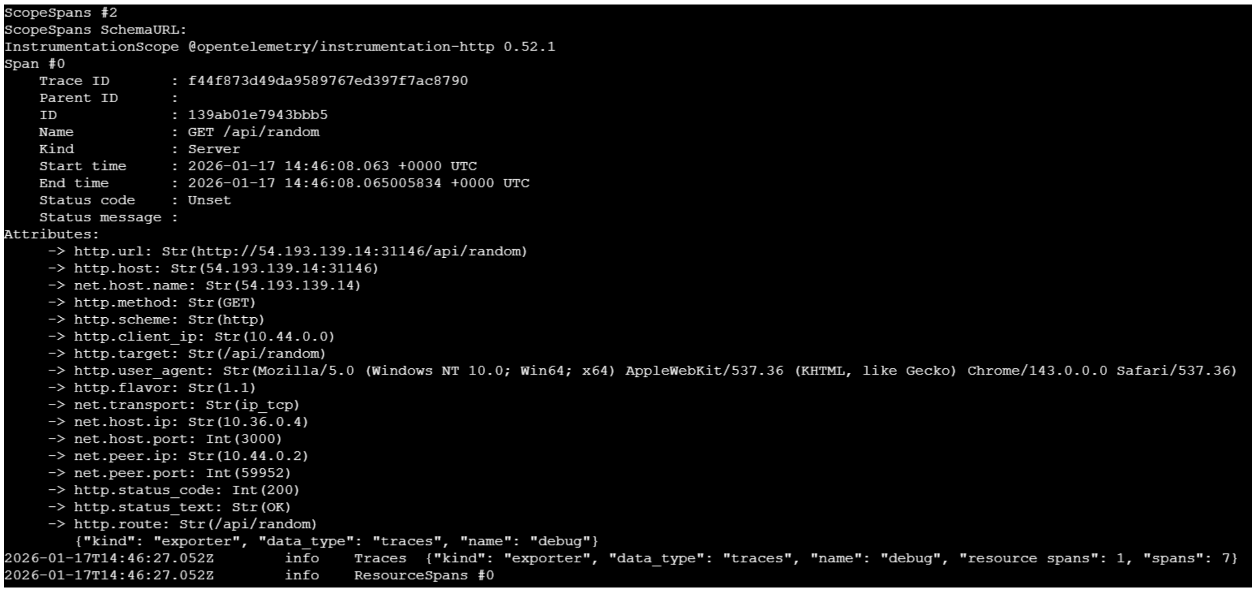
kubectl logs -l app=otel-collector --tail=200

13.2.1 /api/health traces



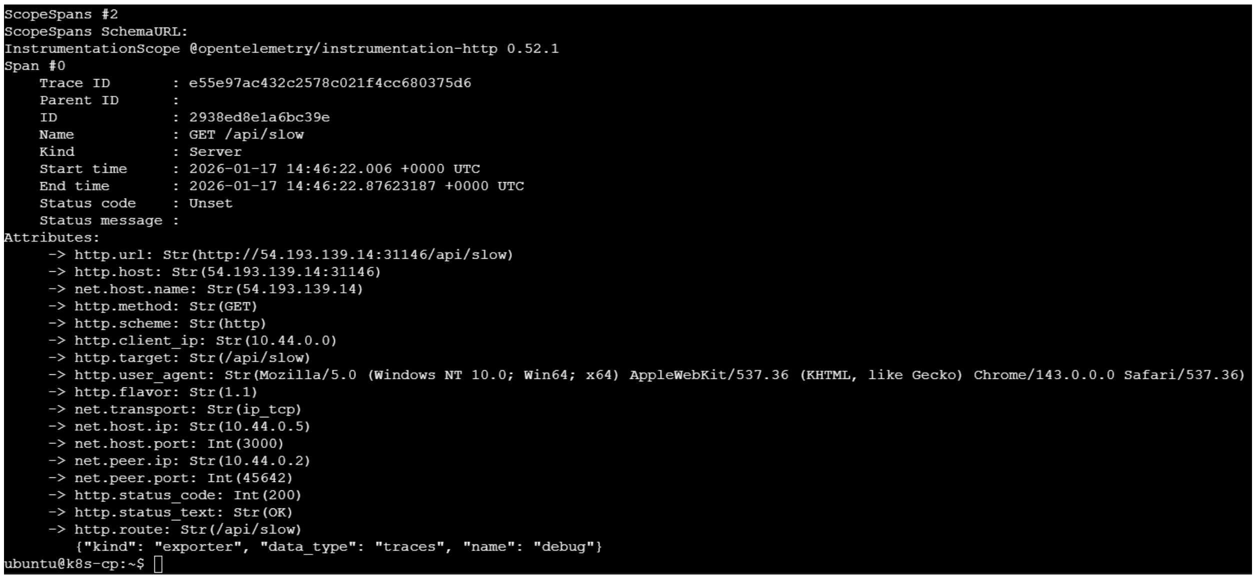
```
Events:
SpanEvent #0
     -> Name: exception
     -> Timestamp: 2026-01-17 14:45:42.449221959 +0000 UTC
     -> DroppedAttributesCount: 0
     -> Attributes::
          -> exception.type: Str(ENOENT)
          -> exception.message: Str(ENOENT: no such file or directory, stat '/app/public/api/health')
          -> exception.stacktrace: Str(Error: ENOENT: no such file or directory, stat '/app/public/api/health')
ScopeSpans #2
ScopeSpans SchemaURL:
InstrumentationScope @opentelemetry/instrumentation-http 0.52.1
Span #0
    Trace ID       : a10d98fad4df80804b9f647c94ee2c17
    Parent ID      :
    ID             : e9e33b8f35882cbb
    Name           : GET /api/health
    Kind           : Server
    Start time     : 2026-01-17 14:45:42.449 +0000 UTC
    End time       : 2026-01-17 14:45:42.450515307 +0000 UTC
    Status code    : Unset
    Status message :
Attributes:
     -> http.url: Str(http://54.193.139.14:31146/api/health)
     -> http.host: Str(54.193.139.14:31146)
     -> net.host.name: Str(54.193.139.14)
     -> http.method: Str(GET)
     -> http.scheme: Str(http)
     -> http.client_ip: Str(10.44.0.0)
     -> http.target: Str(/api/health)
     -> http.user_agent: Str(Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36)
     -> http.flavor: Str(1.1)
     -> net.transport: Str(ip_tcp)
     -> net.host.ip: Str(10.44.0.5)
     -> net.host.port: Int(3000)
     -> net.peer.ip: Str(10.44.0.2)
     -> net.peer.port: Int(33836)
     -> http.status_code: Int(200)
     -> http.status_text: Str(OK)
     -> http.route: Str(/api/health)
        {"kind": "exporter", "data_type": "traces", "name": "debug"}
2026-01-17T14:46:13.220Z        info    Traces  {"kind": "exporter", "data_type": "traces", "name": "debug", "resource spans": 1, "spans": 7}
2026-01-17T14:46:13.220Z        info    ResourceSpans #0
```

13.2.2 /api/random traces



```
ScopeSpans #2
ScopeSpans SchemaURL:
InstrumentationScope @opentelemetry/instrumentation-http 0.52.1
Span #0
    Trace ID       : f44f873d49da9589767ed397f7ac8790
    Parent ID      :
    ID             : 139ab01e7943bbb5
    Name           : GET /api/random
    Kind           : Server
    Start time     : 2026-01-17 14:46:08.063 +0000 UTC
    End time       : 2026-01-17 14:46:08.065005834 +0000 UTC
    Status code    : Unset
    Status message :
Attributes:
     -> http.url: Str(http://54.193.139.14:31146/api/random)
     -> http.host: Str(54.193.139.14:31146)
     -> net.host.name: Str(54.193.139.14)
     -> http.method: Str(GET)
     -> http.scheme: Str(http)
     -> http.client_ip: Str(10.44.0.0)
     -> http.target: Str(/api/random)
     -> http.user_agent: Str(Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36)
     -> http.flavor: Str(1.1)
     -> net.transport: Str(ip_tcp)
     -> net.host.ip: Str(10.36.0.4)
     -> net.host.port: Int(3000)
     -> net.peer.ip: Str(10.44.0.2)
     -> net.peer.port: Int(59952)
     -> http.status_code: Int(200)
     -> http.status_text: Str(OK)
     -> http.route: Str(/api/random)
        {"kind": "exporter", "data_type": "traces", "name": "debug"}
2026-01-17T14:46:27.052Z        info    Traces  {"kind": "exporter", "data_type": "traces", "name": "debug", "resource spans": 1, "spans": 7}
2026-01-17T14:46:27.052Z        info    ResourceSpans #0
```

13.2.3 /api/slow traces



```
ScopeSpans #2
ScopeSpans SchemaURL:
InstrumentationScope @opentelemetry/instrumentation-http 0.52.1
Span #0
    Trace ID       : e55e97ac432c2578c021f4cc680375d6
    Parent ID      :
    ID             : 2938ed8e1a6bc39e
    Name           : GET /api/slow
    Kind           : Server
    Start time     : 2026-01-17 14:46:22.006 +0000 UTC
    End time       : 2026-01-17 14:46:22.87623187 +0000 UTC
    Status code    : Unset
    Status message :
Attributes:
     -> http.url: Str(http://54.193.139.14:31146/api/slow)
     -> http.host: Str(54.193.139.14:31146)
     -> net.host.name: Str(54.193.139.14)
     -> http.method: Str(GET)
     -> http.scheme: Str(http)
     -> http.client_ip: Str(10.44.0.0)
     -> http.target: Str(/api/slow)
     -> http.user_agent: Str(Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/143.0.0.0 Safari/537.36)
     -> http.flavor: Str(1.1)
     -> net.transport: Str(ip_tcp)
     -> net.host.ip: Str(10.44.0.5)
     -> net.host.port: Int(3000)
     -> net.peer.ip: Str(10.44.0.2)
     -> net.peer.port: Int(45642)
     -> http.status_code: Int(200)
     -> http.status_text: Str(OK)
     -> http.route: Str(/api/slow)
        {"kind": "exporter", "data_type": "traces", "name": "debug"}
ubuntu@k8s-cp:~$ 
```

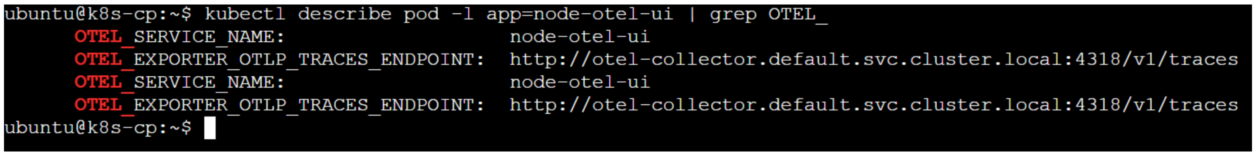
Explanation:

- Confirms that the collector **received and exported spans**.
- Debug exporter prints spans to logs, making it easy to verify end-to-end trace collection.

13.3 Verify OTEL Environment Variables in Node.js Pods

Command:

*kubectl describe pod -l app=node-otel-ui | grep OTEL_*



```
ubuntu@k8s-cp:~$ kubectl describe pod -l app=node-otel-ui | grep OTEL_
      OTEL_SERVICE_NAME:                   node-otel-ui
      OTEL_EXPORTER_OTLP_TRACES_ENDPOINT:  http://otel-collector.default.svc.cluster.local:4318/v1/traces
      OTEL_SERVICE_NAME:                   node-otel-ui
      OTEL_EXPORTER_OTLP_TRACES_ENDPOINT:  http://otel-collector.default.svc.cluster.local:4318/v1/traces
ubuntu@k8s-cp:~$
```

Explanation:

- Confirms OTEL_SERVICE_NAME and OTEL_EXPORTER_OTLP_TRACES_ENDPOINT are set correctly in each pod.
- Ensures spans are exported to the correct collector endpoint i.e. ***http://otel-collector.default.svc.cluster.local:4318/v1/traces***

15. Common Issues & Resolutions

This section documents common issues encountered while deploying and operating a self-managed Kubernetes cluster (kubeadm) on EC2, along with their root causes and resolutions. These scenarios are typical in non-managed Kubernetes environments and highlight important operational lessons.

15.1 OpenTelemetry Collector CrashLoopBackOff

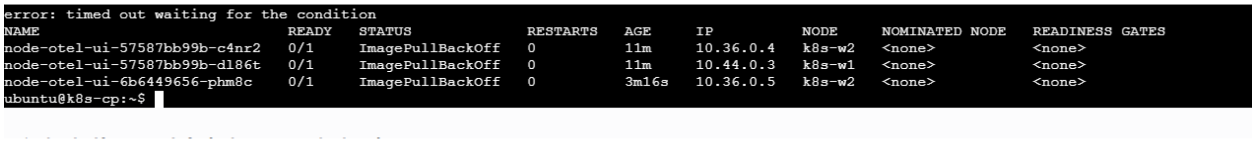
Issue:
The OpenTelemetry Collector pod repeatedly entered a CrashLoopBackOff state shortly after deployment.

Cause:
The collector configuration used the logging exporter, which has been deprecated in newer versions of the OpenTelemetry Collector image. As a result, the collector failed during startup due to an invalid configuration.

Fix:
The ConfigMap was updated to replace the deprecated logging exporter with the supported debug exporter. After applying the updated configuration, the collector Deployment was restarted, and the pod successfully transitioned to a Running state.

15.2 ImagePullBackOff on Node.js UI Pods

Issue:
The Node.js UI application pods failed to start and showed an ImagePullBackOff status.



```
error: timed out waiting for the condition
NAME                            READY   STATUS             RESTARTS   AGE    IP          NODE     NOMINATED NODE   READINESS GATES
node-otel-ui-57587bb99b-c4nr2   0/1     ImagePullBackOff   0          11m    10.36.0.4   k8s-w2   <none>           <none>
node-otel-ui-57587bb99b-dl86t   0/1     ImagePullBackOff   0          11m    10.44.0.3   k8s-w1   <none>           <none>
node-otel-ui-6b6449656-phm8c    0/1     ImagePullBackOff   0          3m16s  10.36.0.5   k8s-w2   <none>           <none>
ubuntu@k8s-cp:~$
```

Cause:
The original container image was hosted in Amazon ECR. In a self-managed kubeadm cluster, worker nodes do not automatically authenticate to ECR using IAM roles, unlike in Amazon EKS. As a result, the nodes lacked the required credentials to pull the image.

Fix:
To simplify deployment and avoid registry authentication complexity, the application image

was pushed to Docker Hub as a public repository (sardarnoor1/node-otel-ui:1.0). The Kubernetes Deployment was updated to reference the Docker Hub image, allowing worker nodes to pull the image successfully.

15.3 HTTPS Error When Accessing Application via NodePort

Issue:
Accessing the application using https://<node-ip>:<NodePort> resulted in a browser SSL/TLS error.

Cause:
The Ingress controller service was exposed using a NodePort without TLS configuration. Since no TLS certificates were configured for the Ingress, the NodePort only served plain HTTP traffic. Attempting HTTPS access caused the SSL handshake to fail.

Fix:
The application was accessed using HTTP instead (http://<node-ip>:<NodePort>), which worked as expected. As a future enhancement, TLS can be enabled by configuring cert-manager and HTTPS-enabled Ingress resources.

Screenshot References:

- **[SS-33]** Browser error when attempting HTTPS access
- **[SS-34]** Application UI successfully loaded over HTTP

16. Final Validation Checklist

This final section confirms that all core Kubernetes components, application workloads, networking, and observability services are functioning correctly. The commands below were executed from the control-plane node to validate the overall cluster health.

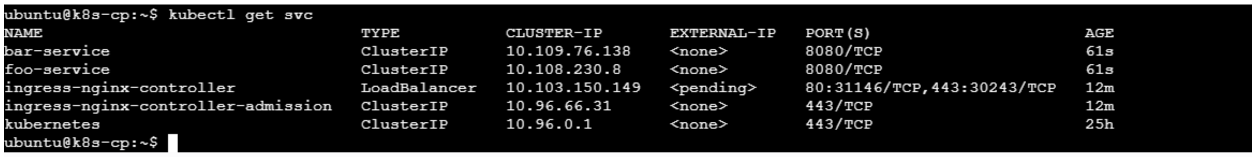
16.1 Service Validation

The following command was run to verify that all required Kubernetes Services were created successfully and exposed as expected:

- *kubectl get svc*

What this confirms:

- Core Kubernetes service (kubernetes) is available.
- Application-related services (foo-service, bar-service) are reachable internally via ClusterIP.
- The Ingress controller service (ingress-nginx-controller) is exposed via **NodePort**, with HTTP traffic mapped to port 31146.
- The EXTERNAL-IP for the LoadBalancer remains <pending>, which is expected in a self-managed kubeadm cluster on EC2.



```
ubuntu@k8s-cp:~$ kubectl get svc
NAME                                 TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
bar-service                          ClusterIP      10.109.76.138    <none>        8080/TCP                     61s
foo-service                          ClusterIP      10.108.230.8     <none>        8080/TCP                     61s
ingress-nginx-controller             LoadBalancer   10.103.150.149   <pending>     80:31146/TCP,443:30243/TCP   12m
ingress-nginx-controller-admission   ClusterIP      10.96.66.31      <none>        443/TCP                      12m
kubernetes                           ClusterIP      10.96.0.1        <none>        443/TCP                      25h
ubuntu@k8s-cp:~$
```

16.2 Pod Validation (Default Namespace)

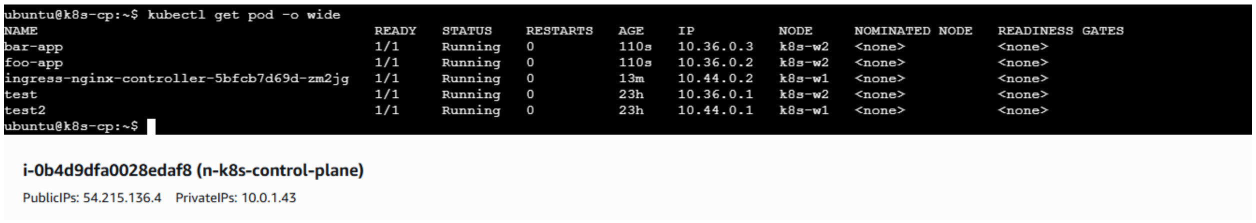
The following command was run to verify that all application pods in the default namespace were running correctly and to inspect their node placement:

- *kubectl get pod -o wide*

What this confirms:

- All application pods (bar-app, foo-app, test, test2) are in Running state with 1/1 ready status.
- The Ingress NGINX controller pod is operational with no restarts.

- Pods are distributed across worker nodes (k8s-w1, k8s-w2) as expected.
- Each pod has been assigned a cluster-internal IP address from the Pod network (10.36.x.x, 10.44.x.x ranges).
- The NOMINATED NODE and READINESS GATES columns show <none>, indicating no special scheduling constraints or custom readiness checks.



```
ubuntu@k8s-cp:~$ kubectl get pod -o wide
NAME                                       READY   STATUS    RESTARTS   AGE    IP          NODE     NOMINATED NODE   READINESS GATES
bar-app                                    1/1     Running   0          110s   10.36.0.3   k8s-w2   <none>           <none>
foo-app                                    1/1     Running   0          110s   10.36.0.2   k8s-w2   <none>           <none>
ingress-nginx-controller-5bfcb7d69d-zm2jg  1/1     Running   0          13m    10.44.0.2   k8s-w1   <none>           <none>
test                                       1/1     Running   0          23h    10.36.0.1   k8s-w2   <none>           <none>
test2                                      1/1     Running   0          23h    10.44.0.1   k8s-w1   <none>           <none>
ubuntu@k8s-cp:~$
```

i-0b4d9dfa0028edaf8 (n-k8s-control-plane)

PublicIPs: 54.215.136.4   PrivateIPs: 10.0.1.43

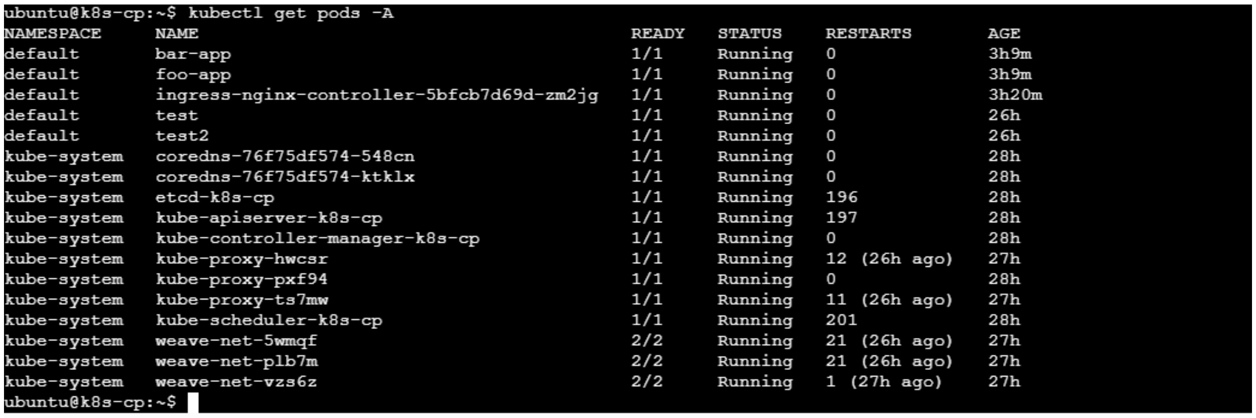
16.3 Pod Validation (All Namespaces)

The following command was run to verify the health of all pods across all namespaces in the cluster:

- *kubectl get pods -A*

What this confirms:

- **default namespace**: Application pods and Ingress controller are running without issues.
- **kube-system namespace**: All core Kubernetes components are operational:
    - CoreDNS pods (coredns-*) are running for DNS resolution.
    - etcd, kube-apiserver, kube-controller-manager, and kube-scheduler are healthy on the control plane.
    - kube-proxy pods are running on all nodes for network routing.
    - Weave Net CNI pods (weave-net-*) are deployed with 2/2 ready status, providing pod networking across the cluster.
- Some pods show restart counts which may indicate previous cluster maintenance or node reboots but do not affect current functionality.
- All pods have appropriate age indicators, confirming cluster stability over time.



```
ubuntu@k8s-cp:~$ kubectl get pods -A
NAMESPACE     NAME                                        READY   STATUS    RESTARTS       AGE
default       bar-app                                     1/1     Running   0              3h9m
default       foo-app                                     1/1     Running   0              3h9m
default       ingress-nginx-controller-5bfcb7d69d-zm2jg   1/1     Running   0              3h20m
default       test                                        1/1     Running   0              26h
default       test2                                       1/1     Running   0              26h
kube-system   coredns-76f75df574-548cn                    1/1     Running   0              28h
kube-system   coredns-76f75df574-ktklx                    1/1     Running   0              28h
kube-system   etcd-k8s-cp                                 1/1     Running   196            28h
kube-system   kube-apiserver-k8s-cp                       1/1     Running   197            28h
kube-system   kube-controller-manager-k8s-cp              1/1     Running   0              28h
kube-system   kube-proxy-hwcsr                            1/1     Running   12 (26h ago)   27h
kube-system   kube-proxy-pxf94                            1/1     Running   0              28h
kube-system   kube-proxy-ts7mw                            1/1     Running   11 (26h ago)   27h
kube-system   kube-scheduler-k8s-cp                       1/1     Running   201            28h
kube-system   weave-net-5wmqf                             2/2     Running   21 (26h ago)   27h
kube-system   weave-net-plb7m                             2/2     Running   21 (26h ago)   27h
kube-system   weave-net-vzs6z                             2/2     Running   1 (27h ago)    27h
ubuntu@k8s-cp:~$ 
```

16) Conclusion

A complete self-managed Kubernetes platform was built on AWS EC2 using kubeadm and Weave Net. A custom Node.js UI application was deployed and exposed externally through NGINX Ingress on a NodePort. OpenTelemetry was integrated end-to-end: the application exported traces over OTLP to an in-cluster OpenTelemetry Collector, and collector logs confirmed trace and span ingestion. This project demonstrates core Kubernetes operations (cluster bootstrapping, deployments, services, ingress routing) and practical observability integration.