Task 16

Kubernetes Cluster on AWS EC2 Using kubeadm with Observability Stack

Umar Satti

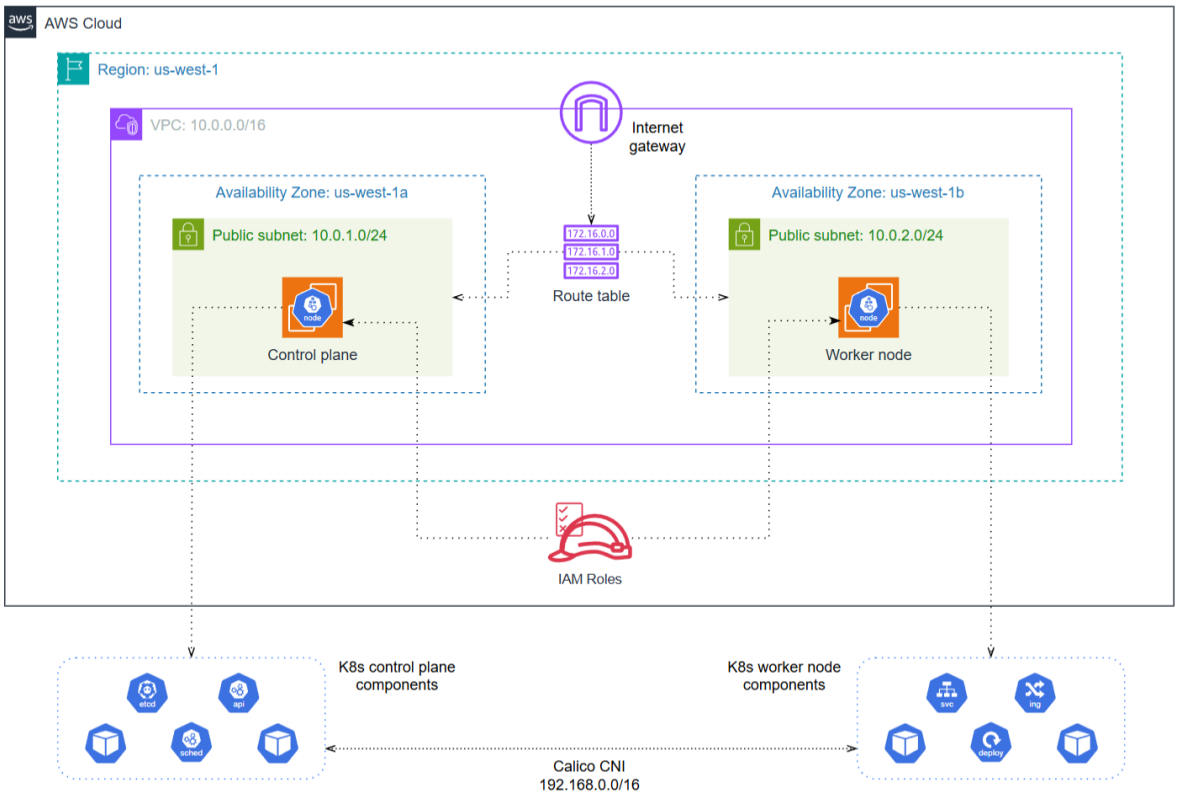
Table of Contents

1. Task Description

This project demonstrates creation of a Kubernetes cluster on AWS EC2 instances using kubeadm, deployed across multiple Availability Zones in us-west-1 with the control plane in us-west-1a and a worker node in us-west-1b.

Kubernetes components are manually configured on Ubuntu EC2 instances with Calico or Flannel for pod networking. A containerized Node.js application is exposed via NGINX Ingress and instrumented using OpenTelemetry, with traces visualized in Jaeger. Prometheus and Grafana, deployed via Helm, provide cluster metrics and monitoring dashboards.

2. Architecture Diagram



3. Virtual Private Cloud (VPC)

3.1 VPC Configuration

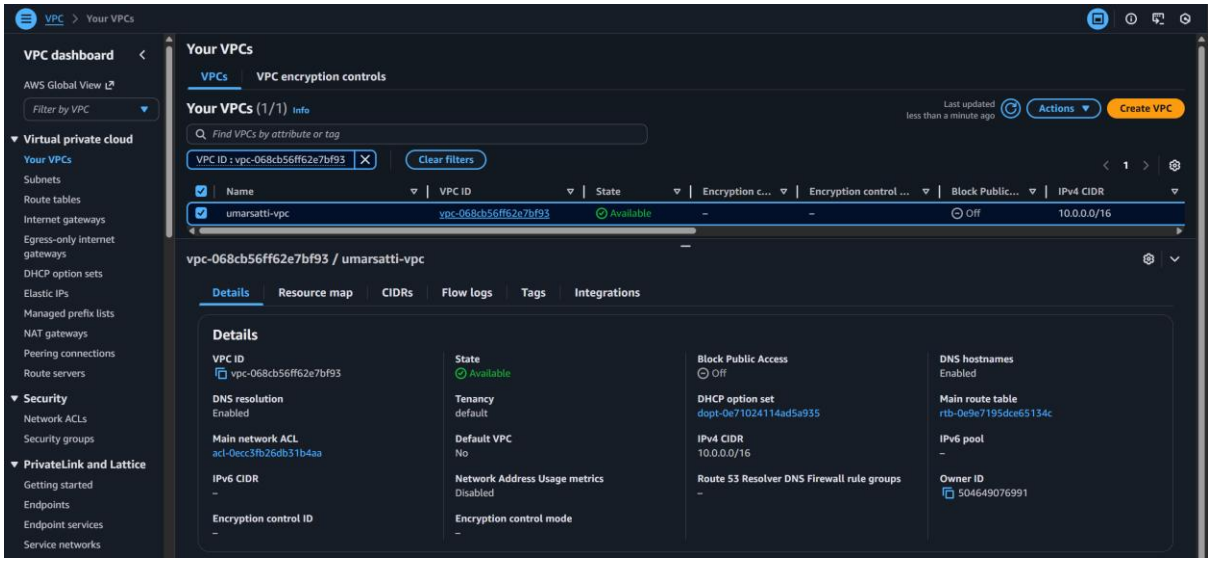
A **Virtual Private Cloud (VPC)** is a logically isolated network environment within AWS, allowing the creation of secure and organized infrastructure.

Steps:

1. Sign in to the AWS Management Console.
2. Navigate to VPC service using the search bar at the top.
3. In the VPC Console, click **Your VPCs.**
4. Click **Create VPC** button on the top right.
5. Choose **VPC only** option for 'Resources to create'.
6. Choose a name for the VPC (example: Umar-VPC) and add an IPv4 CIDR block (example: 10.0.0.0/16).
7. Leave the **Tenancy** as **Default**.

As shown in the image below, under **Details** tab, a VPC has been created with the following configuration:

- **Name:** umarsatti-vpc
- **IPv4 CIDR range:** 10.0.0.0/16
- **VPC ID:** vpc-068cb56ff62e7bf93



3.2 Internet Gateway

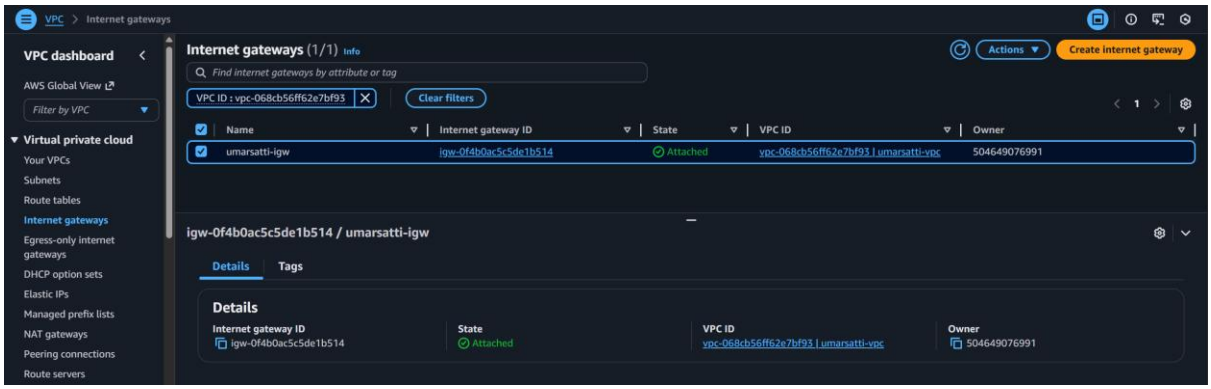
An **Internet Gateway (IGW)** enables communication between instances in the VPC and the public internet.

Steps:

1. Within the VPC Console, click **Internet gateways**.
2. Click on **Create internet gateway** button.
3. Choose a **Name** and add optional Tags**.**
4. Click **Create internet gateway** button. This creates an Internet gateway outside the VPC, and it needs to attach to the VPC created earlier.
5. Select **Actions** on the top right and click **Attach to VPC**. Choose the VPC ID (created earlier) and click **Attach internet gateway**. This attaches the Internet gateway to the VPC, allowing internet access.

As shown in the image below under **Details** tab, an internet gateway has been created with the following configuration:

- **Name:** umarsatti-igw
- **State:** Attached
- **VPC ID:** vpc-068cb56ff62e7bf93
- **Internet gateway ID:** igw-0f4b0ac5c5de1b514

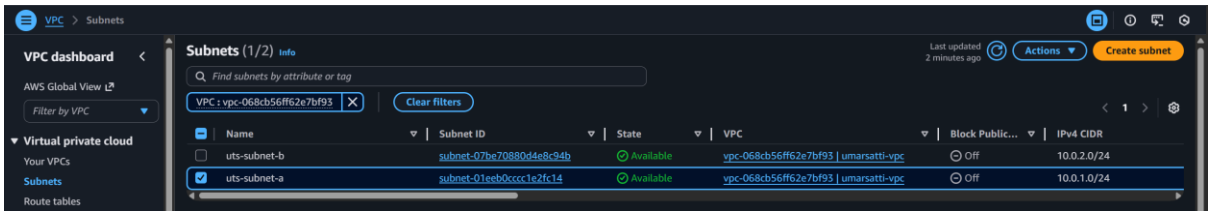


3.3 Subnets

Steps:

1. Within the VPC Console, click **Subnets**.
2. Click on **Create subnet** button.
3. Choose a **Subnet name, Availability Zone, IPv4 VPC CIDR block, IPv4 subnet CIDR Block** and optional **Tags.**
4. After adding the required subnets, click on **Create subnet** button at the bottom right.

As shown in the image below under **Subnets** section, two subnets have been created with the following configuration:



Public Subnet A:

- **Name:** uts-subnet-a
- **Subnet ID:** subnet-01eeb0cccc1e2fc14
- **IPv4 CIDR:** 10.0.1.0/24

Public Subnet B:

- **Name:** uts-subnet-b
- **Subnet ID:** subnet-07be70880d4e8c94b
- **IPv4 CIDR:** 10.0.2.0/24

3.4 Route Tables

Route Tables define how network traffic moves within the VPC. Each route table directs traffic either internally (within the VPC) or externally (to the internet or a NAT gateway).

Steps:

1. Within the VPC Console, click **Route tables**.
2. Click on **Create route table** button on the top right.
3. Choose a **Name, VPC**, and optional **Tags.**
4. When done, click on **Create route table** button to create the resource.

As shown in the image below under **Subnets** section, two route tables have been created with the following configuration:

Public Route table A:

- **Name:** uts-routetable-a
- **Route table ID:** rtb-05b9bb4820905f422

Public Route table B:

- **Name:** uts-routetable-b
- **Route table ID:** rtb-066474442ca344cf5

3.5 Setting up Routes for Route tables

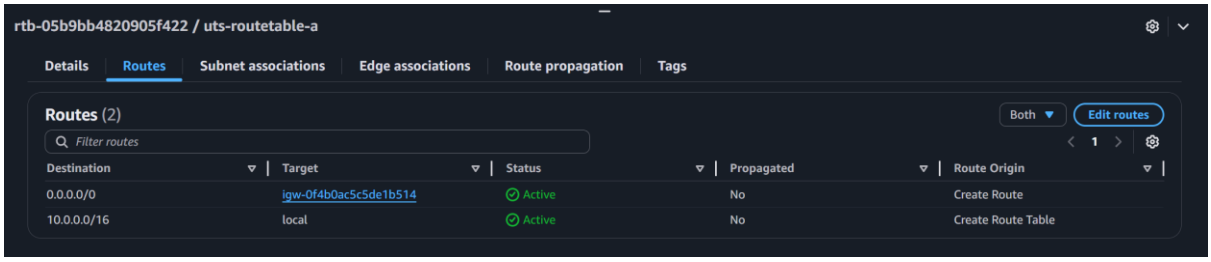
Steps:

1. Within the Route tables console, select a Route table and click **Edit routes** on the right.
2. Add a new route by choosing a **Destination** and **Target**.
3. Click **Save changes**. This adds a route to this route table.

Public Routes

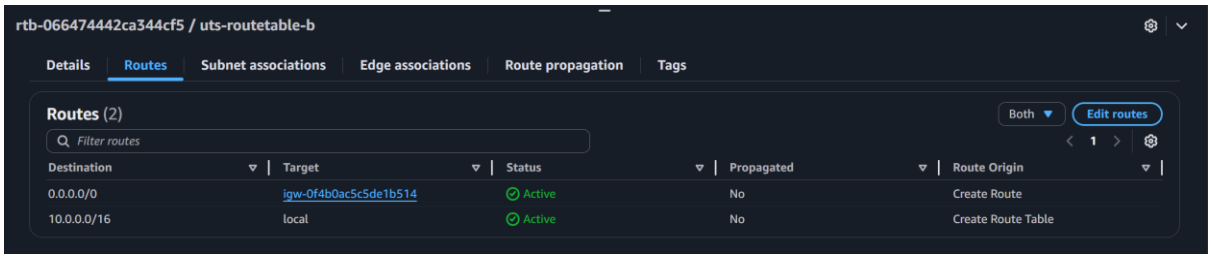
1. uts-routetable-a:

- Destination: **0.0.0.0/0**
- Target: **umarsatti-igw | igw-0f4b0ac5c5de1b514** (Internet gateway)



2. uts-routetable-b:

- Destination: **0.0.0.0/0**
- Target: **umarsatti-igw | igw-0f4b0ac5c5de1b514** (Internet gateway)



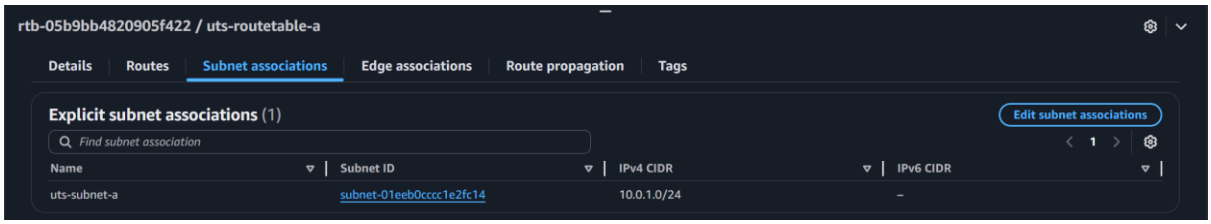
3.6 Route Table Association

Steps:

1. Within the Route tables console, select a Route table and click **Subnet associations** tab under the details section.
2. Click **Edit subnet associations** and select the specific subnet
3. Click **Save associations** button on the bottom right.

Route table A assoication

- **Names:** uts-subnet-a
- **Subnet ID:** subnet-01eeb0cccc1e2fc14
- **IPv4 CIDR:** 10.0.1.0/24



Route table B assoication

- **Name:** uts-subnet-b
- **Subnet ID:** subnet-07be70880d4e8c94b
- **IPv4 CIDR:** 10.0.2.0/24

3.7 VPC Resource Map

A VPC Resource Map is a visual diagram in the AWS console that shows the VPC's architecture, including all its components (subnets, route tables, gateways, etc.) and their interconnections.

3.8 Security Groups

Security Groups act as virtual firewalls that regulate inbound and outbound traffic at the instance level.

Steps:

1. Within the VPC Console, click **Security Groups**.
2. Click on **Create security group** button on the top right.
3. Choose a **Security group name**, **Description**, and **VPC** (umarsatti-vpc).
4. Add Inbound rules by selecting **Type, Protocol, Port range, Source,** and **Description** (optional).
5. Leave Outbound rules as default (All traffic).

3.8.1 Control plane security group

Here is the configuration for the public subnet security group:

- Security group name: **k8s-control-plane-s3**
- Test environment so a lot of security group inbound rules are open:

Type	Protocol	Port range	Source
HTTP	TCP	80	0.0.0.0/0
SSH	TCP	22	My IP
Custom	TCP	30000 - 32767	0.0.0.0/0
Custom	TCP	4317 - 4318	
Custom	TCP	6443	
Custom	TCP	2379 - 2380	
Custom	TCP	10250	

3.8.2 Worker node security group

Here is the configuration for the public subnet security group:

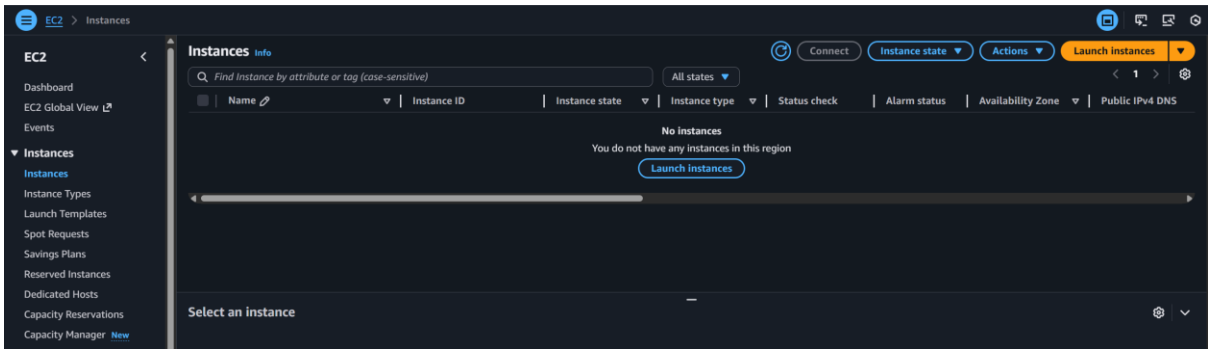
- Security group name: **k8s-worker-node-sg**
- Test environment so a lot of security group inbound rules are open:

Type	Protocol	Port range	Source
HTTP	TCP	80	0.0.0.0/0
SSH	TCP	22	My IP
Custom	TCP	30000 - 32767	0.0.0.0/0
Custom	TCP	10250	

4. Set up EC2 Instances

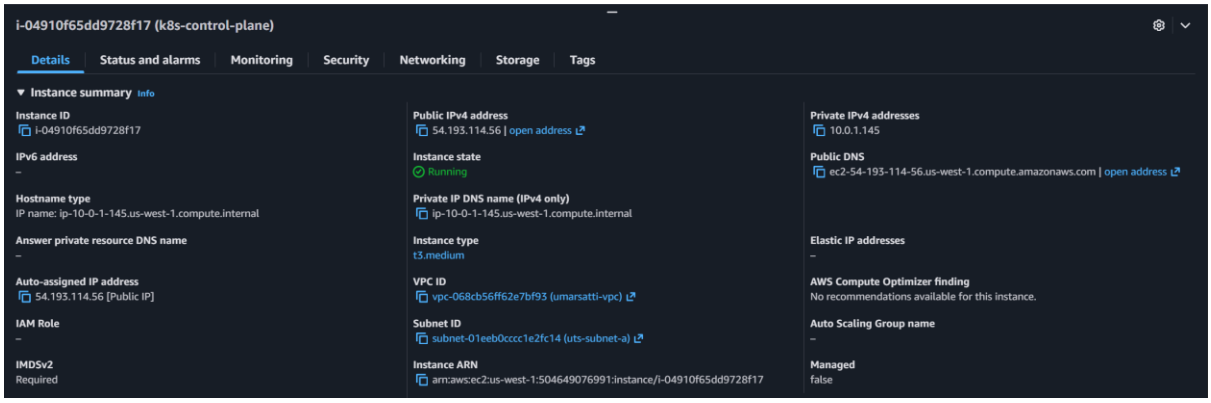
Steps:

1. In the EC2 Console, click **Instances** on the left navigation panel.
2. Click **Launch Instances** button on the top right.
3. Choose a **Name, OS Image, AMI, Architecture, Instance type,** and **Key pair**.
4. Edit **Network settings.**
5. Choose **VPC, Subnet, Availability Zone, Auto-assign public IP, security group, storage, and IAM instance profile.**
6. Leave the rest as default and click **Launch Instance.**



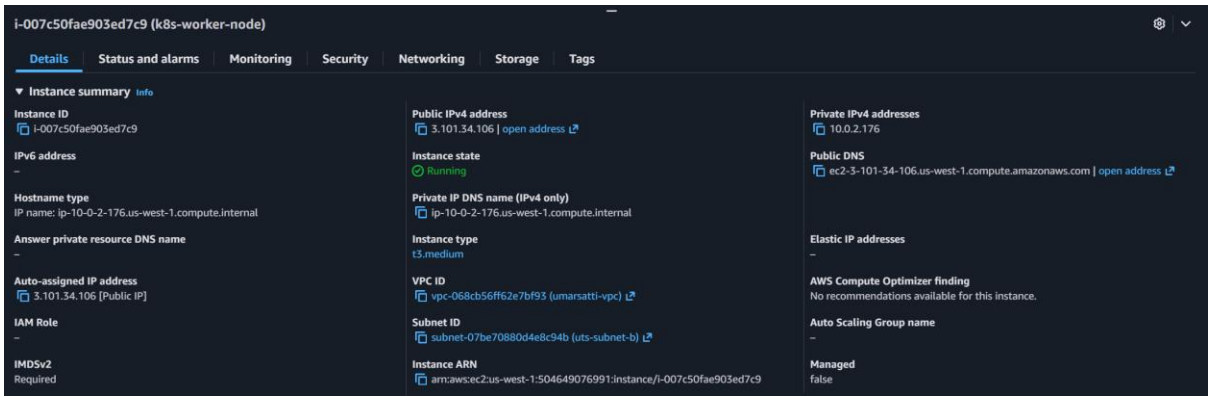
Control Plane EC2:

- Name: k8s-control-plane
- AMI: Ubuntu 24.04
- Architecture: 64-bit (x86)
- Instance type: t3.medium
- Key pair: uts.pem (for SSH login)
- VPC: umarsatti-vpc
- Subnet: uts-public-a
- Availability zone: us-west-1a
- Security group name: k8s-control-plane-sg
- Storage: 20 GiB gp3
- **IAM Instance profile** should have **SSMManagedInstanceCore** and **S3FullAccess** policies



Worker Node EC2:

- Name: k8s-worker-node
- AMI: Ubuntu 24.04
- Architecture: 64-bit (x86)
- Instance type: t3.medium
- Key pair: uts.pem (for SSH login)
- VPC: umarsatti-vpc
- Subnet: uts-public-b
- Availability zone: us-west-1b
- Security group name: k8s-worker-node-sg
- Storage: 20 GiB gp3
- **IAM Instance profile** should have **SSMManagedInstanceCore** and **S3FullAccess** policies



5. Set up Kubernetes cluster on EC2 using kubeadm

The following link provides a complete guide on how to boostrap clusters using kubeadm on Linux operating systems.

https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/

5.1 On Control Plane and Worker Node

1. Check required ports

Before setting up the Kubernetes cluster, verify that the necessary network ports are accessible on both control plane and worker nodes. Kubernetes components communicate through specific TCP and UDP ports, and blocking these ports will prevent proper cluster operation.

Control plane

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	6443	Kubernetes API server	All
TCP	Inbound	2379-2380	etcd server client API	kube-apiserver, etcd
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10259	kube-scheduler	Self
TCP	Inbound	10257	kube-controller-manager	Self

Although etcd ports are included in control plane section, you can also host your own etcd cluster externally or on custom ports.

Worker node(s)

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10256	kube-proxy	Self, Load balancers
TCP	Inbound	30000-32767	NodePort Services†	All
UDP	Inbound	30000-32767	NodePort Services†	All

2. Set hostnames (optional)

Setting descriptive hostnames for kubernetes nodes helps identify them easily within the cluster.

- **hostnamectl set-hostname**: To assign meaningful names like "control-plane" and "worker-node" to EC2 instances.
- **exec bash**: To apply the hostname change to the current shell session.

```
ubuntu@ip-10-0-1-229:~$ sudo hostnamectl set-hostname control-plane
ubuntu@ip-10-0-1-229:~$ exec bash
ubuntu@control-plane:~$
```

```
ubuntu@ip-10-0-2-251:~$ sudo hostnamectl set-hostname worker-node
ubuntu@ip-10-0-2-251:~$ exec bash
ubuntu@worker-node:~$
```

3. Swap configuration

Kubernetes requires swap memory to be disabled on all nodes. The kubelet (Kubernetes node agent) will fail to start if swap is enabled, as Kubernetes needs predictable memory management for pod scheduling and resource allocation.

- Use **sudo swapoff -a** command to immediately disable swap on the system.

```
ubuntu@control-plane:~$ sudo swapoff -a
ubuntu@control-plane:~$
```

```
ubuntu@worker-node:~$ sudo swapoff -a
ubuntu@worker-node:~$
```

4. Enable required Kernel modules (for networking)

Kubernetes networking requires specific Linux kernel modules to function properly. The configuration file **/etc/modules-load.d/k8s.conf** lists essential modules:

- **overlay:** For container filesystem layers
- **br_netfilter:** For bridge network filtering
- **iptables:** For network traffic routing and filtering

- **modprobe:** To load these modules immediately and ensure that CNI and kube-proxy can manage pod networking correctly

```
ubuntu@control-plane:~$ cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
ip_tables
iptable_filter
iptable_nat
iptable_mangle
iptable_raw
iptable_security
EOF

sudo modprobe overlay
sudo modprobe br_netfilter
sudo modprobe ip_tables
sudo modprobe iptable_filter
sudo modprobe iptable_nat
sudo modprobe iptable_mangle
sudo modprobe iptable_raw
sudo modprobe iptable_security
overlay
br_netfilter
ip_tables
iptable_filter
iptable_nat
iptable_mangle
iptable_raw
iptable_security
ubuntu@control-plane:~$
```

```
ubuntu@worker-node:~$ cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
ip_tables
iptable_filter
iptable_nat
iptable_mangle
iptable_raw
iptable_security
EOF

sudo modprobe overlay
sudo modprobe br_netfilter
sudo modprobe ip_tables
sudo modprobe iptable_filter
sudo modprobe iptable_nat
sudo modprobe iptable_mangle
sudo modprobe iptable_raw
sudo modprobe iptable_security
overlay
br_netfilter
ip_tables
iptable_filter
iptable_nat
iptable_mangle
iptable_raw
iptable_security
ubuntu@worker-node:~$
```

5. Configure kernel parameters (sysctl) for pod networking and packet forwarding

Kubernetes requires specific kernel settings to allow pods to communicate across network interfaces and to properly handle packet forwarding. Configure the following parameters:

- **net.bridge.bridge-nf-call-iptables = 1**
  - Allows iptables to process bridged IPv4 traffic
- **net.bridge.bridge-nf-call-ip6tables = 1**

    - Allows iptables to process bridged IPv6 traffic
- **net.ipv4.ip_forward = 1**
    - Enables packet forwarding between network interfaces

After configuring the parameters, apply changes immediately and persist them across reboots using:

- **sudo sysctl –system**
- Verify using: ***sysctl net.ipv4.ip_forward***

```
ubuntu@control-plane:~$ cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
EOF
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
ubuntu@control-plane:~$ sudo sysctl --system
* Applying /usr/lib/sysctl.d/10-apparmor.conf ...
* Applying /etc/sysctl.d/10-bufferbloat.conf ...
* Applying /etc/sysctl.d/10-console-messages.conf ...
* Applying /etc/sysctl.d/10-ipv6-privacy.conf ...
* Applying /etc/sysctl.d/10-kernel-hardening.conf ...
* Applying /etc/sysctl.d/10-magic-sysrq.conf ...
* Applying /etc/sysctl.d/10-map-count.conf ...
* Applying /etc/sysctl.d/10-network-security.conf ...
* Applying /etc/sysctl.d/10-ptrace.conf ...
* Applying /etc/sysctl.d/10-zeropage.conf ...
* Applying /etc/sysctl.d/50-cloudimg-settings.conf ...
* Applying /usr/lib/sysctl.d/50-pid-max.conf ...
* Applying /etc/sysctl.d/99-cloudimg-ipv6.conf ...
* Applying /usr/lib/sysctl.d/99-protect-links.conf ...
* Applying /etc/sysctl.d/99-sysctl.conf ...
* Applying /etc/sysctl.d/k8s.conf ...
* Applying /etc/sysctl.conf ...
kernel.apparmor_restrict_unprivileged_userns = 1
net.core.default_qdisc = fq_codel
kernel.printk = 4 4 1 7
net.ipv6.conf.all.use_tempaddr = 2
```

```
ubuntu@worker-node:~$ cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
EOF
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
ubuntu@worker-node:~$ sudo sysctl --system
* Applying /usr/lib/sysctl.d/10-apparmor.conf ...
* Applying /etc/sysctl.d/10-bufferbloat.conf ...
* Applying /etc/sysctl.d/10-console-messages.conf ...
* Applying /etc/sysctl.d/10-ipv6-privacy.conf ...
* Applying /etc/sysctl.d/10-kernel-hardening.conf ...
* Applying /etc/sysctl.d/10-magic-sysrq.conf ...
* Applying /etc/sysctl.d/10-map-count.conf ...
* Applying /etc/sysctl.d/10-network-security.conf ...
* Applying /etc/sysctl.d/10-ptrace.conf ...
* Applying /etc/sysctl.d/10-zeropage.conf ...
* Applying /etc/sysctl.d/50-cloudimg-settings.conf ...
* Applying /usr/lib/sysctl.d/50-pid-max.conf ...
* Applying /etc/sysctl.d/99-cloudimg-ipv6.conf ...
* Applying /usr/lib/sysctl.d/99-protect-links.conf ...
* Applying /etc/sysctl.d/99-sysctl.conf ...
* Applying /etc/sysctl.d/k8s.conf ...
* Applying /etc/sysctl.conf ...
kernel.apparmor_restrict_unprivileged_userns = 1
net.core.default_qdisc = fq_codel
kernel.printk = 4 4 1 7
net.ipv6.conf.all.use_tempaddr = 2
```

6. Installation and configuration containerd runtime

Containerd is the container runtime used by Kubernetes to manage the container lifecycle.

i. Installation steps

- Install required dependencies including ca-certificates, curl, gnupg, lsb-release
- Install the **containerd** package

```
ubuntu@control-plane:~$ sudo apt-get install -y ca-certificates curl gnupg lsb-release
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
ca-certificates is already the newest version (20240203).
ca-certificates set to manually installed.
curl is already the newest version (8.5.0-2ubuntu10.6).
curl set to manually installed.
gnupg is already the newest version (2.4.4-2ubuntu17.3).
gnupg set to manually installed.
lsb-release is already the newest version (12.0-2).
lsb-release set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 68 not upgraded.
ubuntu@control-plane:~$ sudo apt-get install -y containerd
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
```

```
ubuntu@worker-node:~$ sudo apt-get install -y ca-certificates curl gnupg lsb-release
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
ca-certificates is already the newest version (20240203).
ca-certificates set to manually installed.
curl is already the newest version (8.5.0-2ubuntu10.6).
curl set to manually installed.
gnupg is already the newest version (2.4.4-2ubuntu17.3).
gnupg set to manually installed.
lsb-release is already the newest version (12.0-2).
lsb-release set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 68 not upgraded.
ubuntu@worker-node:~$ sudo apt-get install -y containerd
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
```

ii. Configuration steps

- Generate the default configuration file: **/etc/containerd/config.toml**
- Update the configuration by setting **SystemdCgroup = true**. This ensures that containerd uses system for cgroup management

```
ubuntu@control-plane:~$ sudo mkdir -p /etc/containerd
ubuntu@control-plane:~$ sudo containerd config default | sudo tee /etc/containerd/config.toml
```

```
ubuntu@worker-node:~$ sudo mkdir -p /etc/containerd
ubuntu@worker-node:~$ sudo containerd config default | sudo tee /etc/containerd/config.toml
```

```
ubuntu@control-plane:~$ sudo sed -i 's/SystemdCgroup = false/SystemdCgroup = true/g' /etc/containerd/config.toml
ubuntu@control-plane:~$ cat /etc/containerd/config.toml | grep SystemdCgroup
            SystemdCgroup = true
ubuntu@control-plane:~$
```

```
ubuntu@worker-node:~$ sudo sed -i 's/SystemdCgroup = false/SystemdCgroup = true/g' /etc/containerd/config.toml
ubuntu@worker-node:~$ cat /etc/containerd/config.toml | grep SystemdCgroup
            SystemdCgroup = true
ubuntu@worker-node:~$
```

iii. Finalize setup

- Restart the **containerd** service to apply changes
- Enable **containerd** to start automatically on boot

```
ubuntu@control-plane:~$ sudo systemctl restart containerd
ubuntu@control-plane:~$ sudo systemctl enable containerd
ubuntu@control-plane:~$ sudo systemctl status containerd
● containerd.service - containerd container runtime
     Loaded: loaded (/usr/lib/systemd/system/containerd.service; enabled; preset: enabled)
     Active: active (running) since Tue 2026-01-06 21:20:10 UTC; 18s ago
       Docs: https://containerd.io
   Main PID: 1834 (containerd)
      Tasks: 7
     Memory: 13.3M (peak: 13.7M)
        CPU: 99ms
     CGroup: /system.slice/containerd.service
             └─1834 /usr/bin/containerd

Jan 06 21:20:10 control-plane containerd[1834]: time="2026-01-06T21:20:10.997075122Z" level=info msg="Start subscribing>
Jan 06 21:20:10 control-plane containerd[1834]: time="2026-01-06T21:20:10.997181676Z" level=info msg=serving... address>
Jan 06 21:20:10 control-plane containerd[1834]: time="2026-01-06T21:20:10.997221576Z" level=info msg="Start recovering >
Jan 06 21:20:10 control-plane containerd[1834]: time="2026-01-06T21:20:10.997257495Z" level=info msg=serving... address>
Jan 06 21:20:10 control-plane containerd[1834]: time="2026-01-06T21:20:10.997297911Z" level=info msg="Start event monit>
Jan 06 21:20:10 control-plane containerd[1834]: time="2026-01-06T21:20:10.997311838Z" level=info msg="Start snapshots s>
Jan 06 21:20:10 control-plane containerd[1834]: time="2026-01-06T21:20:10.997326049Z" level=info msg="Start cni network>
Jan 06 21:20:10 control-plane containerd[1834]: time="2026-01-06T21:20:10.997336319Z" level=info msg="Start streaming s>
Jan 06 21:20:10 control-plane systemd[1]: Started containerd.service - containerd container runtime.
Jan 06 21:20:10 control-plane containerd[1834]: time="2026-01-06T21:20:10.997942773Z" level=info msg="containerd succes>
```

```
ubuntu@worker-node:~$ sudo systemctl restart containerd
ubuntu@worker-node:~$ sudo systemctl enable containerd
ubuntu@worker-node:~$ sudo systemctl status containerd
● containerd.service - containerd container runtime
     Loaded: loaded (/usr/lib/systemd/system/containerd.service; enabled; preset: enabled)
     Active: active (running) since Tue 2026-01-06 21:20:12 UTC; 47s ago
       Docs: https://containerd.io
   Main PID: 1849 (containerd)
      Tasks: 8
     Memory: 13.5M (peak: 14.0M)
        CPU: 143ms
     CGroup: /system.slice/containerd.service
             └─1849 /usr/bin/containerd

Jan 06 21:20:12 worker-node containerd[1849]: time="2026-01-06T21:20:12.818529559Z" level=info msg="Start subscribing c>
Jan 06 21:20:12 worker-node containerd[1849]: time="2026-01-06T21:20:12.818601199Z" level=info msg="Start recovering st>
Jan 06 21:20:12 worker-node containerd[1849]: time="2026-01-06T21:20:12.818676024Z" level=info msg="Start event monitor"
Jan 06 21:20:12 worker-node containerd[1849]: time="2026-01-06T21:20:12.818690175Z" level=info msg=serving... address=/>
Jan 06 21:20:12 worker-node containerd[1849]: time="2026-01-06T21:20:12.818695306Z" level=info msg="Start snapshots syn>
Jan 06 21:20:12 worker-node containerd[1849]: time="2026-01-06T21:20:12.818720390Z" level=info msg="Start cni network c>
Jan 06 21:20:12 worker-node containerd[1849]: time="2026-01-06T21:20:12.818732909Z" level=info msg="Start streaming ser>
Jan 06 21:20:12 worker-node containerd[1849]: time="2026-01-06T21:20:12.818753559Z" level=info msg=serving... address=/>
Jan 06 21:20:12 worker-node containerd[1849]: time="2026-01-06T21:20:12.818814242Z" level=info msg="containerd successf>
Jan 06 21:20:12 worker-node systemd[1]: Started containerd.service - containerd container runtime.
```

7. Install Kubernetes components (kubelet, kubeadm, and kubectl)

Install the core Kubernetes components on all nodes in the cluster. Start by updating the package list and installing the required prerequisites needed to access the Kubernetes repository. Pre-requisites include:

- apt-transport-https
- ca-certificates
- curl
- gpg

```
ubuntu@control-plane:~$ sudo apt-get update
Hit:1 http://us-west-1.ec2.archive.ubuntu.com/ubuntu noble InRelease
Hit:2 http://us-west-1.ec2.archive.ubuntu.com/ubuntu noble-updates InRelease
Hit:3 http://us-west-1.ec2.archive.ubuntu.com/ubuntu noble-backports InRelease
Hit:4 http://security.ubuntu.com/ubuntu noble-security InRelease
Reading package lists... Done
ubuntu@control-plane:~$ sudo apt-get install -y apt-transport-https ca-certificates curl gpg
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
```

```
ubuntu@worker-node:~$ sudo apt-get update
Hit:1 http://us-west-1.ec2.archive.ubuntu.com/ubuntu noble InRelease
Hit:2 http://us-west-1.ec2.archive.ubuntu.com/ubuntu noble-updates InRelease
Hit:3 http://us-west-1.ec2.archive.ubuntu.com/ubuntu noble-backports InRelease
Hit:4 http://security.ubuntu.com/ubuntu noble-security InRelease
Reading package lists... Done
ubuntu@worker-node:~$ sudo apt-get install -y apt-transport-https ca-certificates curl gpg
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
```

Next, add the official Kubernetes apt repository by downloading and adding the GPG key, then registering the repository in the system's sources list.

```
ubuntu@worker-node:~$ ls /etc/apt/keyrings
ubuntu@worker-node:~$ sudo mkdir -p -m 755 /etc/apt/keyrings
ubuntu@worker-node:~$
```

```
ubuntu@control-plane:~$ ls /etc/apt/keyrings
ubuntu@control-plane:~$ sudo mkdir -p -m 755 /etc/apt/keyrings
ubuntu@control-plane:~$
```

```
ubuntu@control-plane:~$ curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.34/deb/Release.key | sudo gpg --dearmor -o /etc/
apt/keyrings/kubernetes-apt-keyring.gpg
ubuntu@control-plane:~$ echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/sta
ble:/v1.34/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.34/deb/ /
ubuntu@control-plane:~$
```

```
ubuntu@worker-node:~$ curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.34/deb/Release.key | sudo gpg --dearmor -o /etc/ap
t/keyrings/kubernetes-apt-keyring.gpg
ubuntu@worker-node:~$ echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stabl
e:/v1.34/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.34/deb/ /
ubuntu@worker-node:~$
```

After the repository is configured, install the Kubernetes components:

- **kubelet**: node-level agent responsible for running pods
- **kubeadm**: tool used to bootstrap and manage the cluster
- **kubectl**: command-line interface for Kubernetes administration

```
ubuntu@worker-node:~$ sudo apt-get update
Hit:1 http://us-west-1.ec2.archive.ubuntu.com/ubuntu noble InRelease
Hit:2 http://us-west-1.ec2.archive.ubuntu.com/ubuntu noble-updates InRelease
Hit:3 http://us-west-1.ec2.archive.ubuntu.com/ubuntu noble-backports InRelease
Get:4 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.34/deb  InRelease [1227 B]
Hit:5 http://security.ubuntu.com/ubuntu noble-security InRelease
Get:6 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.34/deb  Packages [6868 B]
Fetched 8095 B in 1s (13.4 kB/s)
Reading package lists... Done
ubuntu@worker-node:~$ sudo apt-get install -y kubelet kubeadm kubectl
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
```

```
ubuntu@control-plane:~$ sudo apt-get update
Hit:1 http://us-west-1.ec2.archive.ubuntu.com/ubuntu noble InRelease
Hit:2 http://us-west-1.ec2.archive.ubuntu.com/ubuntu noble-updates InRelease
Hit:3 http://us-west-1.ec2.archive.ubuntu.com/ubuntu noble-backports InRelease
Get:4 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.34/deb  InRelease [1227 B]
Hit:5 http://security.ubuntu.com/ubuntu noble-security InRelease
Get:6 https://prod-cdn.packages.k8s.io/repositories/isv:/kubernetes:/core:/stable:/v1.34/deb  Packages [6868 B]
Fetched 8095 B in 1s (14.1 kB/s)
Reading package lists... Done
ubuntu@control-plane:~$ sudo apt-get install -y kubelet kubeadm kubectl
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
```

Once installed, prevent unintended version upgrades by placing the packages on hold using apt-mark hold kubelet kubeadm kubectl, ensuring consistent versions across all cluster nodes.

```
ubuntu@control-plane:~$ sudo apt-mark hold kubelet kubeadm kubectl
kubelet set on hold.
kubeadm set on hold.
kubectl set on hold.
ubuntu@control-plane:~$
```

```
ubuntu@worker-node:~$ sudo apt-mark hold kubelet kubeadm kubectl
kubelet set on hold.
kubeadm set on hold.
kubectl set on hold.
ubuntu@worker-node:~$
```

8. Verification of Kubernetes components installation

Verify that the Kubernetes components were installed correctly by checking their versions using the appropriate commands. Run the following commands:

- kubeadm version
- kubectl version –client

Next, verify the kubelet service installation by checking its status with **systemctl status kubelet**. At this stage, the service may appear inactive or dead, which is expected until the cluster has been initialized.

Finally, enable the kubelet service to ensure it starts automatically after cluster initialization:

- systemctl enable --now kubelet

```
ubuntu@control-plane:~$ kubeadm version
kubeadm version: &version.Info{Major:"1", Minor:"34", EmulationMajor:"", EmulationMinor:"", MinCompatibilityMajor:"", Mi
nCompatibilityMinor:"", GitVersion:"v1.34.3", GitCommit:"df11db1c0f08fab3c0baee1e5ce6efbf816af7f1", GitTreeState:"clean"
, BuildDate:"2025-12-09T15:05:15Z", GoVersion:"go1.24.11", Compiler:"gc", Platform:"linux/amd64"}
ubuntu@control-plane:~$ service kubelet status
○ kubelet.service - kubelet: The Kubernetes Node Agent
     Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled; preset: enabled)
    Drop-In: /usr/lib/systemd/system/kubelet.service.d
             └─10-kubeadm.conf
     Active: inactive (dead)
       Docs: https://kubernetes.io/docs/
ubuntu@control-plane:~$ kubectl version --client
Client Version: v1.34.3
Kustomize Version: v5.7.1
ubuntu@control-plane:~$ sudo systemctl enable --now kubelet
ubuntu@control-plane:~$
```

```
ubuntu@worker-node:~$ kubeadm version
kubeadm version: &version.Info{Major:"1", Minor:"34", EmulationMajor:"", EmulationMinor:"", MinCompatibilityMajor:"", Mi
nCompatibilityMinor:"", GitVersion:"v1.34.3", GitCommit:"df11db1c0f08fab3c0baee1e5ce6efbf816af7f1", GitTreeState:"clean"
, BuildDate:"2025-12-09T15:05:15Z", GoVersion:"go1.24.11", Compiler:"gc", Platform:"linux/amd64"}
ubuntu@worker-node:~$ service kubelet status
○ kubelet.service - kubelet: The Kubernetes Node Agent
     Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled; preset: enabled)
    Drop-In: /usr/lib/systemd/system/kubelet.service.d
             └─10-kubeadm.conf
     Active: inactive (dead)
       Docs: https://kubernetes.io/docs/
ubuntu@worker-node:~$ kubectl version --client
Client Version: v1.34.3
Kustomize Version: v5.7.1
ubuntu@worker-node:~$ sudo systemctl enable --now kubelet
ubuntu@worker-node:~$
```

5.2 Installation on Control Plane

1. Initialize kubeadm and set up a Pod Network

Initialize the Kubernetes control plane using **kubeadm init**, specifying the pod network CIDR with the **--pod-network-cidr** flag. The CIDR value must match the Container Network Interface (CNI) plugin used. Common pod network CIDRs:

- 192.168.0.0/16 for **Calico**
- 10.244.0.0/16 for **Flannel**

```
ubuntu@control-plane:~$ sudo kubeadm init --pod-network-cidr=192.168.0.0/16
I0106 21:34:01.884321    3126 version.go:260] remote version is much newer: v1.35.0; falling back to: stable-1.34
[init] Using Kubernetes version: v1.34.3
```

After the control plane initializes successfully, configure **kubectl** access by copying the admin configuration file to **$HOME/.kube/config** and setting the appropriate ownership and permissions. This allows the current user to interact with the cluster using **kubectl** command.

```
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/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 10.0.1.229:6443 --token mxnqmy.7hxa6xs3jziuz9hx \
        --discovery-token-ca-cert-hash sha256:c4448d91657b39d8e949e43167e482a3874a1d5912babe1f826669a073d9c3cf
ubuntu@control-plane:~$ mkdir -p $HOME/.kube
ubuntu@control-plane:~$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
ubuntu@control-plane:~$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
ubuntu@control-plane:~$
```

The initialization output also includes a **kubeadm join** command containing a **token** and **discovery hash**. This command will be used later to add worker nodes to the cluster.

2. Install Container Network Interface (CNI)

Install a Container Network Interface (CNI) plugin to enable pod-to-pod communication across the cluster. A commonly used option is Calico, which provides both networking and network policy enforcement.

Calico details:

- Uses the 192.168.0.0/16 pod CIDR
- Requires **TCP port 5473** to be open between nodes
- For Flannel CNI, use --pod-network-cidr=10.244.0.0/16

Flannel details:

- Uses the 10.244.0.0/16 pod CIDR
- Requires **UDP port 8472** to be open between nodes

```
ubuntu@control-plane:~$ kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.31.3/manifests/tiger
a-operator.yaml
namespace/tigera-operator created
serviceaccount/tigera-operator created
clusterrole.rbac.authorization.k8s.io/tigera-operator-secrets created
clusterrole.rbac.authorization.k8s.io/tigera-operator created
clusterrolebinding.rbac.authorization.k8s.io/tigera-operator created
rolebinding.rbac.authorization.k8s.io/tigera-operator-secrets created
deployment.apps/tigera-operator created
ubuntu@control-plane:~$ kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.31.3/manifests/custo
m-resources.yaml
installation.operator.tigera.io/default created
apiserver.operator.tigera.io/default created
goldmane.operator.tigera.io/default created
whisker.operator.tigera.io/default created
```

Deploy the CNI by applying the manifests provided by the CNI vendor using **kubectl create -f** with the appropriate YAML files. Once applied, verify the installation by ensuring all pods in the calico-system namespace are running. This includes the Calico API server, controllers, node agents, and CSI components.

```
ubuntu@control-plane:~$ kubectl get pods -n calico-system
NAME                                       READY   STATUS    RESTARTS   AGE
calico-apiserver-77d4f9f884-lfzql          1/1     Running   0          42m
calico-apiserver-77d4f9f884-w558s          1/1     Running   0          42m
calico-kube-controllers-799b7d56b4-jdnmf   1/1     Running   0          42m
calico-node-4jgnc                          1/1     Running   0          7m26s
calico-node-5mrpr                          1/1     Running   0          7m26s
calico-typha-6bd9b7688-7lfsd               1/1     Running   0          42m
csi-node-driver-7znk5                      2/2     Running   0          42m
csi-node-driver-qhq2x                      2/2     Running   0          42m
goldmane-8b794fbc6-25dwz                   1/1     Running   0          42m
whisker-bfc56c85d-cv6sc                    2/2     Running   0          42m
ubuntu@control-plane:~$
```

5.3 Configuration on Worker Nodes

Join worker nodes to the cluster

On each worker node, join the cluster using the **kubeadm join** command generated during control plane initialization. This command contains the control plane IP address and **API server port (6443)**, along with a bootstrap token and the discovery token CA certificate hash used for secure authentication.

When the command is executed, the **kubelet** service starts automatically and performs pre-flight checks, retrieves the cluster configuration, and establishes a secure TLS connection with the control plane. After the process is completed successfully, the worker node is registered with the cluster and becomes available to schedule workloads.

```
ubuntu@worker-node:~$ sudo kubeadm join 10.0.1.229:6443 --token mxnqmy.7hxa6xs3jziuz9hx \
        --discovery-token-ca-cert-hash sha256:c4448d91657b39d8e949e43167e482a3874a1d5912babe1f826669a073d9c3cf
[preflight] Running pre-flight checks
[preflight] Reading configuration from the "kubeadm-config" ConfigMap in namespace "kube-system"...
[preflight] Use 'kubeadm init phase upload-config kubeadm --config your-config-file' to re-upload it.
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/instance-config.yaml"
[patches] Applied patch of type "application/strategic-merge-patch+json" to target "kubeletconfiguration"
[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-check] Waiting for a healthy kubelet at http://127.0.0.1:10248/healthz. This can take up to 4m0s
[kubelet-check] The kubelet is healthy after 1.50195618s
[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.
```

5.4 Verification (using Control Plane)

Verify the overall cluster state from the control plane using **kubectl** commands.

kubectl get nodes

Displays all nodes in the cluster, including control plane and worker nodes. Each node should show a **Ready** status once fully configured.

```
ubuntu@control-plane:~$ kubectl get nodes
NAME            STATUS   ROLES           AGE   VERSION
control-plane   Ready    control-plane   46h   v1.34.3
worker-node     Ready    <none>          46h   v1.34.3
ubuntu@control-plane:~$
```

kubectl get pods -A

Lists all pods across all namespaces to confirm that core system components (such as kube-system and calico-system) and the CNI plugin are running correctly. All pods should be in the **Running** state with the expected number of ready containers.

6. Application Deployment and Monitoring

The control plane node has administrative access to the Kubernetes cluster through the configured **kubectl** context (admin.conf). It is responsible for managing cluster-wide resources such as namespaces, Custom Resource Definitions (CRDs), controllers, and operators.

Worker nodes do not have **kubectl** configured by default and are intended only to run application workloads, not to perform cluster administration tasks.

Note: From this point onward, all commands **must be executed on the Control Plane** EC2 instance.

6.1 cert-manager

cert-manager is required for the **OpenTelemetry Operator** because it automates the creation and management of TLS certificates used by Kubernetes admission webhooks and operator components. Deploy cert-manager by applying the official manifest from the GitHub release, which installs the required CRDs and creates the cert-manager namespace.

Install cert-manager:

- kubectl apply -f https://github.com/cert-manager/cert-manager/releases/download/v1.19.2/cert-manager.yaml

```
ubuntu@control-plane:~$ kubectl apply -f https://github.com/cert-manager/cert-manager/releases/download/v1.19.2/cert-man
ager.yaml
namespace/cert-manager created
customresourcedefinition.apiextensions.k8s.io/challenges.acme.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/orders.acme.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/certificaterequests.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/certificates.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/clusterissuers.cert-manager.io created
customresourcedefinition.apiextensions.k8s.io/issuers.cert-manager.io created
```

Verify the installation:

- kubectl get pods --namespace cert-manager

Ensure that the following pods are running:

- cert-manager
- cert-manager-cainjector
- cert-manager-webhook

```
ubuntu@control-plane:~$ kubectl get pods --namespace cert-manager
NAME                                       READY   STATUS    RESTARTS   AGE
cert-manager-79559475b4-x2mnp              1/1     Running   0          55s
cert-manager-cainjector-966fc8fbc-wthxk    1/1     Running   0          55s
cert-manager-webhook-854cf5f458-cbflj      1/1     Running   0          55s
ubuntu@control-plane:~$
```

6.2 OpenTelemetry

OpenTelemetry enables observability by collecting metrics, logs, and traces from applications running in the Kubernetes cluster.

1. Install the OpenTelemetry Operator

Before installing the operator, ensure that **cert-manager** is successfully running. Apply the **OpenTelemetry** Operator manifest to deploy the operator components and required CRDs.

Install the operator:

- kubectl apply -f https://github.com/open-telemetry/opentelemetry-operator/releases/latest/download/opentelemetry-operator.yaml

Notes:

- This creates the **opentelemetry-operator-system** namespace
- Installs CRDs for managing OpenTelemetry resources

Verify namespaces:

- kubectl get ns / kubectl get namespace

2. Creating an OpenTelemetry Collector resource

Create a dedicated namespace for OpenTelemetry components and deploy an OpenTelemetry Collector using a custom YAML configuration file. The collector defines how telemetry data is received, processed, and exported.

Create the namespace:

- kubectl create namespace opentelemetry

Create and apply the collector configuration:

- vim otel-collector.yaml
- kubectl apply -f otel-collector.yaml

Verify the deployment:

- kubectl get pods -n opentelemetry
- kubectl get opentelemetrycollector -n opentelemetry

```
ubuntu@control-plane:~$ kubectl apply -f otel-collector.yaml
opentelemetrycollector.opentelemetry.io/otel-collector created
ubuntu@control-plane:~$ kubectl get pods -n opentelemetry
NAME                                READY   STATUS    RESTARTS   AGE
otel-collector-collector-7c8fc4c6dd-kkjqd   1/1     Running   0          30s
ubuntu@control-plane:~$ kubectl get opentelemetrycollector -n opentelemetry
NAME             MODE         VERSION   READY   AGE   IMAGE                                                                                    MANAGEMENT
otel-collector   deployment   0.141.0   1/1     67s   ghcr.io/open-telemetry/opentelemetry-collector-releases/opentelemetry-collector:0.141.0   managed
ubuntu@control-plane:~$
```

Confirm that the collector pods are running and that the OpenTelemetryCollector custom resource has been successfully created.

6.3 Node.js Application Deployment

This section describes how to deploy a containerized Node.js application to the Kubernetes cluster using **Deployment**, **Service**, and **Ingress** resources. The application is packaged as a Docker image, pushed to DockerHub, and then deployed to the cluster using Kubernetes manifest files.

6.3.1 Node.js application and DockerHub

Create a basic Node.js application in a project directory containing the necessary source code and configuration files. It contains the following:

- Dockerfile
- app.js
- package.json
- index.html

Once the application is ready, authenticate to **DockerHub** from the terminal and build the container image locally. After building the image, tag it with your DockerHub **username** and push it to a **public** DockerHub repository, which allows Kubernetes to pull the image during pod creation.

DockerHub Login:

- docker login -u <Username>
- Enter DockerHub password when prompted

Build, tag, and push the image:

- docker build -t node-app:latest .
- docker tag node-app:latest <Username>/node-app:latest
- docker push <Username>/node-app:latest

Note: The Docker image reference (<Username>/node-app:latest) is specified in the deployment.yaml manifest and is used by Kubernetes to pull the image when creating pods.

6.3.2 Deployment, Service, and Ingress Setup

To expose the Node.js application externally, an **NGINX Ingress Controller** is required. The controller handles incoming HTTP/HTTPS traffic and routes requests to the appropriate Kubernetes services.

```
ubuntu@control-plane:~$ mkdir node-app
ubuntu@control-plane:~$ cd node-app/
ubuntu@control-plane:~/node-app$ vim deployment.yaml
ubuntu@control-plane:~/node-app$ vim service.yaml
ubuntu@control-plane:~/node-app$ vim ingress.yaml
ubuntu@control-plane:~/node-app$ ls -la
total 20
drwxrwxr-x 2 ubuntu ubuntu 4096 Jan  6 23:20 .
drwxr-x--- 6 ubuntu ubuntu 4096 Jan  6 23:20 ..
-rw-rw-r-- 1 ubuntu ubuntu  333 Jan  6 23:20 deployment.yaml
-rw-rw-r-- 1 ubuntu ubuntu  366 Jan  6 23:20 ingress.yaml
-rw-rw-r-- 1 ubuntu ubuntu  167 Jan  6 23:20 service.yaml
ubuntu@control-plane:~/node-app$
```

1. Install NGINX Ingress Controller

Install the NGINX Ingress Controller by applying its official manifest. This creates the required resources in the ingress-nginx namespace.

Install the controller:

- kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.14.0/deploy/static/provider/baremetal/deploy.yaml

Verify the installation:

- kubectl get pods -n ingress-nginx
- kubectl get svc -n ingress-nginx

The **ingress-nginx-controller** service is exposed as a NodePort, allowing external traffic to reach the cluster through worker nodes.

2. Create and apply application manifests

Deploy the application using three Kubernetes manifest files:

Deployment (deployment.yaml)
Defines the Node.js application pod specification and replica count.

- kubectl apply -f deployment.yaml
- kubectl get deployment

Service (service.yaml)
Exposes the deployment internally within the cluster using a ClusterIP service.

- kubectl apply -f service.yaml
- kubectl get svc

Ingress (ingress.yaml)
Configures external routing rules that direct traffic from Ingress controller to the service.

- kubectl apply -f ingress.yaml

- kubectl get ingress

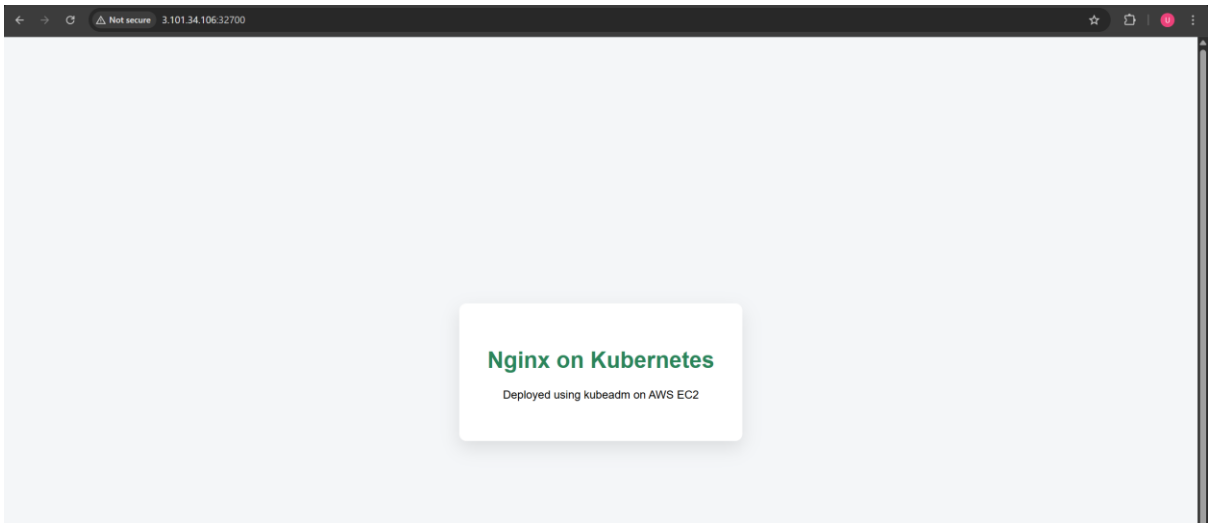
```
ubuntu@control-plane:~/node-app$ kubectl apply -f deployment.yaml
deployment.apps/node-app created
ubuntu@control-plane:~/node-app$ kubectl apply -f service.yaml
service/node-app-service created
ubuntu@control-plane:~/node-app$ kubectl apply -f ingress.yaml
ingress.networking.k8s.io/node-app-ingress created
ubuntu@control-plane:~/node-app$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.14.0/deploy/static/provider/baremetal/d
eploy.yaml
namespace/ingress-nginx created
serviceaccount/ingress-nginx created
serviceaccount/ingress-nginx-admission created
role.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
configmap/ingress-nginx-controller created
service/ingress-nginx-controller created
service/ingress-nginx-controller-admission created
deployment.apps/ingress-nginx-controller created
job.batch/ingress-nginx-admission-create created
job.batch/ingress-nginx-admission-patch created
ingressclass.networking.k8s.io/nginx created
validatingwebhookconfiguration.admissionregistration.k8s.io/ingress-nginx-admission created
ubuntu@control-plane:~/node-app$
```

3. Access the application

The application can be accessed through the **NodePort** exposed by the ingress-nginx-controller service. By default, this is a high-numbered port (for example, 32700) exposed on all worker nodes.

```
ubuntu@control-plane:~$ kubectl get svc -n ingress-nginx
NAME                                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE
ingress-nginx-controller             NodePort    10.96.246.35    <none>        80:32700/TCP,443:31878/TCP   45h
ingress-nginx-controller-admission   ClusterIP   10.103.60.17    <none>        443/TCP                      45h
ubuntu@control-plane:~$
```

- **Example URL**: http://<Worker-Node-IP>:<Node-Port>
- **NodePort** is 32700



When a request is made to this URL, the Ingress controller receives the traffic and routes it to the appropriate service and backend application pods.

6.4 Instrumentation

Instrumentation enables automatic observability for the Node.js application by instructing the OpenTelemetry Operator to inject the OpenTelemetry SDK into application pods at startup.

The **Instrumentation Custom Resource (CR)** tells the OpenTelemetry Operator to:

- Automatically inject the **Node.js OpenTelemetry SDK**
- Set required environment variables (such as OTEL_EXPORTER_OTLP_ENDPOINT)
- Attach instrumentation only to pods matching specific labels
- Export telemetry data to the configured OpenTelemetry Collector
- Apply instrumentation **only when pods start**

Important: Because instrumentation is injected at pod startup time, existing pods must be restarted after the Instrumentation resource is applied. This is handled by re-deploying the application.

6.4.1 Create an Instrumentation resource

Create a file named instrumentation.yaml that defines how the application should be instrumented and where telemetry data should be sent.

Key configuration points:

- **matchLabels.app: node-app**: Matches the labels defined in the Node.js Deployment pods
- **otel-collector.opentelemetry.svc**: Uses the Kubernetes service DNS name of the OpenTelemetry Collector
- **Port 4317**: Specifies the OTLP gRPC endpoint
- **OTEL_SERVICE_NAME**: Defines the logical service name that appears in traces and metrics

This configuration ensures that only the intended application pods are instrumented and that telemetry data is routed correctly.

6.4.2 Apply the Instrumentation resource

Apply the Instrumentation manifest to the cluster and verify that it has been created successfully.

Apply the resource:

- kubectl apply -f instrumentation.yaml

Verify:

- kubectl get instrumentation -n opentelemetry

```
ubuntu@control-plane:~$ kubectl get instrumentation -n opentelemetry
NAME                     AGE   ENDPOINT                                                SAMPLER   SAMPLER ARG
nodejs-instrumentation   45h   http://otel-collector-collector.opentelemetry.svc:4317
ubuntu@control-plane:~$
```

6.4.3 Re-deploy application resources

After applying instrumentation, **re-deploy all application resources** so that the instrumentation is injected during pod startup.

Re-deploy resources in the **opentelemetry** namespace:

- Deployment
- Service
- Ingress

Note: Ensure that all YAML manifest files include the correct namespace (it should be **namespace: opentelemetry** field. Once the pods restart, the OpenTelemetry SDK will be injected automatically, and telemetry data will begin flowing to the OpenTelemetry Collector.

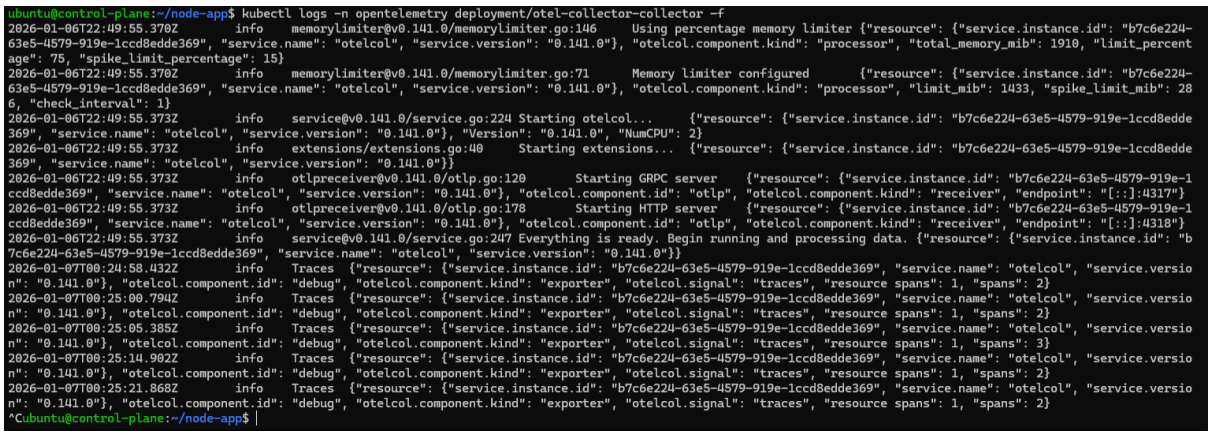
6.5 Jaeger Tracing

After verifying that telemetry data is successfully reaching the OpenTelemetry Collector, the next step is to visualize and analyze distributed traces using **Jaeger UI**. Jaeger provides an intuitive interface for exploring request flows, span durations, and service dependencies.

Verify telemetry is arriving at the OpenTelemetry Collector

Before deploying Jaeger, confirm that the OpenTelemetry Collector is receiving trace data from the Node.js application. Run the following command on the control plane:

- kubectl logs -n opentelemetry deployment/otel-collector-collector



The Collector logs confirm telemetry ingestion and processing:

- **Resource Spans: 1** indicates that the node-app service is successfully exporting trace data.
- **Spans: 2 / 3** represent individual operations, such as incoming HTTP requests and internal middleware execution.
- Debug exporter output verifies that traces are being received and processed according to the configured pipeline.

This confirms that instrumentation and export configuration are working correctly.

6.5.1 Add Jaeger UI for traces

Jaeger is deployed using the **all-in-one** setup, which bundles the Jaeger Collector, Query service, and UI into a single deployment. This approach is well-suited for development and demonstration environments.

A jaeger.yaml manifest is created containing:

- A **Deployment** running the jaegertracing/all-in-one image
- A **Service** of type NodePort to expose Jaeger externally
- Use **kubectl apply -f jaeger.yaml**

Once deployed, the OpenTelemetry Collector is updated to export traces to Jaeger by adding a **Jaeger exporter** to the otel-collector.yaml trace pipeline. After applying the updated manifests, Jaeger begins receiving trace data from the Collector.

Verify the Jaeger service:

- **kubectl get svc -n opentelemetry jaeger**

```
ubuntu@control-plane:~$ kubectl get svc -n opentelemetry jaeger
NAME     TYPE       CLUSTER-IP       EXTERNAL-IP   PORT(S)                          AGE
jaeger   NodePort   10.110.110.153   <none>        16686:30827/TCP,4317:31190/TCP   45h
ubuntu@control-plane:~$
```

Key ports:

- **16686**: Jaeger UI
- **NodePort (e.g., 30827)**: External access to the UI
- **4317**: OTLP gRPC endpoint used by the Collector

Note: Ensure the required NodePort is allowed in the EC2 security group.

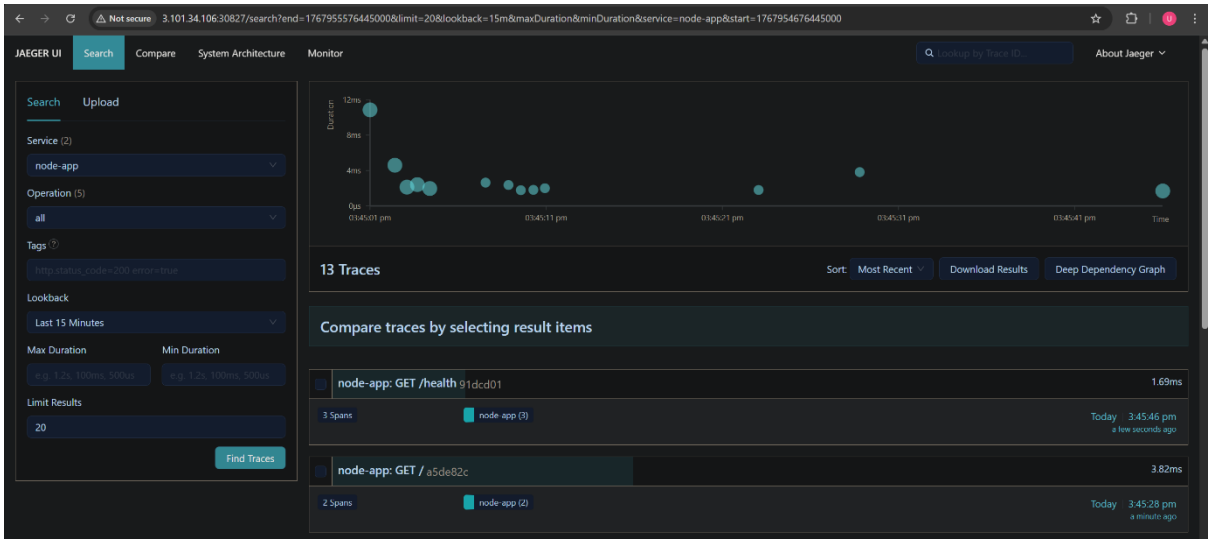
1. Accessing the Jaeger UI

Open the Jaeger UI in a browser using: http://<Worker-Node-IP>:<NodePort>

- Example: http://3.101.34.106:30827

Once accessed, select the node-app service to explore trace data.

2. Jaeger Dashboard

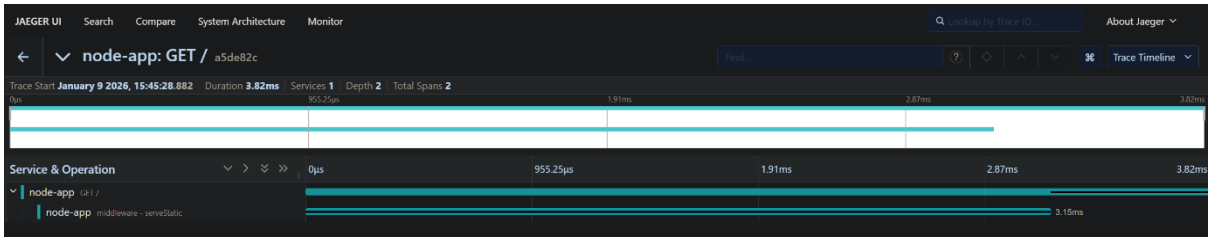


The main dashboard displays all collected traces for the node-app service. It shows:

- Trace count over time
- Request duration distribution
- Individual trace entries sorted by recency

This view helps confirm that requests are actively being traced.

3. GET / trace view

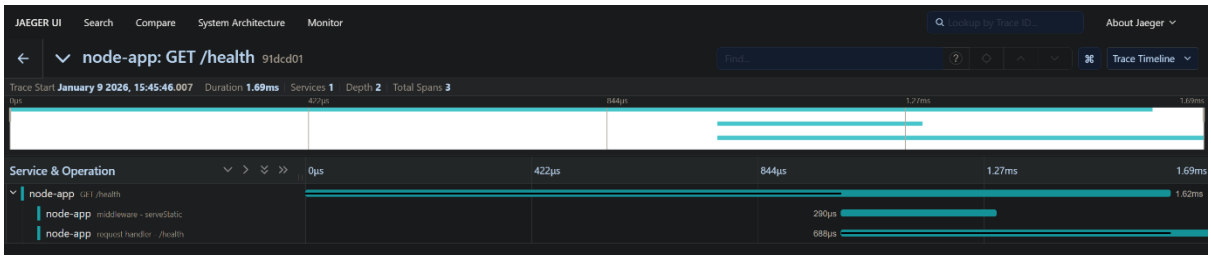


Selecting a trace for the GET / endpoint reveals:

- End-to-end request duration
- Span hierarchy showing request handling and middleware execution
- Timing breakdown for each span

This provides insight into how requests flow through the application.

4. GET /health trace view



The GET /health trace shows a shorter execution path with fewer spans, reflecting the lightweight nature of health-check endpoints. Span duration and depth confirm minimal processing overhead.

6.6 Helm Installation

Helm is a package manager for Kubernetes that simplifies application deployment by using reusable and configurable **charts**. It is commonly used to install complex applications such as monitoring and observability stacks.

Install Helm using the official installation script provided by the Helm project. The script downloads the appropriate binary for your system and configures it automatically.

Download the installation script:

- curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-4

Make the script executable and run it:

- chmod 700 get_helm.sh
- ./get_helm.sh

Verify the installation:

- helm version

```
ubuntu@control-plane:~$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-4
ubuntu@control-plane:~$ chmod 700 get_helm.sh
ubuntu@control-plane:~$ ./get_helm.sh
Downloading https://get.helm.sh/helm-v4.0.4-linux-amd64.tar.gz
Verifying checksum... Done.
Preparing to install helm into /usr/local/bin
helm installed into /usr/local/bin/helm
ubuntu@control-plane:~$ helm version
version.BuildInfo{Version:"v4.0.4", GitCommit:"8650e1dad9e6ae38b41f60b712af9218a0d8cc11", GitTreeState:"clean", GoVersio
n:"go1.25.5", KubeClientVersion:"v1.34"}
ubuntu@control-plane:~$
```

A successful version output confirms that Helm is installed and ready to manage Kubernetes applications and chart repositories.

6.7 Prometheus and Grafana for Metrics and Visualization

Prometheus and Grafana are used together to collect, store, and visualize metrics from the Kubernetes cluster and running workloads.

1. Prometheus Helm Repository

Add the Prometheus Community Helm repository to access the latest monitoring charts, including the kube-prometheus-stack.

- helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
- helm repo update

```
ubuntu@control-plane:~$ helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
"prometheus-community" has been added to your repositories
ubuntu@control-plane:~$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "prometheus-community" chart repository
Update Complete. ⎈Happy Helming!⎈
ubuntu@control-plane:~$
```

2. Install Prometheus + Grafana (kube-prometheus-stack)

The **kube-prometheus-stack** chart deploys a complete monitoring solution, including:

- Prometheus (metrics collection and storage)
- Grafana (visualization)
- Alertmanager (alert handling)
- Node exporters and Kubernetes metrics exporters

Create a dedicated namespace to isolate monitoring components:

- kubectl create namespace monitoring

Install the kube-prometheus-stack:

- helm install kube-prometheus-stack prometheus-community/kube-prometheus-stack --namespace monitoring

```
ubuntu@control-plane:~$ helm install kube-prometheus-stack prometheus-community/kube-prometheus-stack --namespace monitoring
NAME: kube-prometheus-stack
LAST DEPLOYED: Wed Jan  7 10:37:31 2026
NAMESPACE: monitoring
STATUS: deployed
REVISION: 1
DESCRIPTION: Install complete
NOTES:
kube-prometheus-stack has been installed. Check its status by running:
  kubectl --namespace monitoring get pods -l "release=kube-prometheus-stack"

Get Grafana 'admin' user password by running:

  kubectl --namespace monitoring get secrets kube-prometheus-stack-grafana -o jsonpath="{.data.admin-password}" | base64 -d ; e
cho

Access Grafana local instance:

  export POD_NAME=$(kubectl --namespace monitoring get pod -l "app.kubernetes.io/name=grafana,app.kubernetes.io/instance=kube-p
rometheus-stack" -oname)
  kubectl --namespace monitoring port-forward $POD_NAME 3000

Get your grafana admin user password by running:

  kubectl get secret --namespace monitoring -l app.kubernetes.io/component=admin-secret -o jsonpath="{.items[0].data.admin-pass
word}" | base64 --decode ; echo


Visit https://github.com/prometheus-operator/kube-prometheus for instructions on how to create & configure Alertmanager and Pro
metheus instances using the Operator.
ubuntu@control-plane:~$
```

To access Grafana externally, configure it as a **NodePort** service during installation:

- helm install kube-prometheus-stack prometheus-community/kube-prometheus-stack --namespace monitoring --set grafana.service.type=NodePort

Verify the Grafana service:

- kubectl get svc -n monitoring kube-prometheus-stack-grafana

Verify that all monitoring components are running:

- kubectl get pods -n monitoring

```
ubuntu@control-plane:~$ kubectl get pods -n monitoring
NAME                                                     READY   STATUS    RESTARTS   AGE
alertmanager-kube-prometheus-stack-alertmanager-0        2/2     Running   0          2m20s
kube-prometheus-stack-grafana-9d9fdcc45-vxfvn            3/3     Running   0          2m28s
kube-prometheus-stack-kube-state-metrics-669dcf4b9-thb8g 1/1     Running   0          2m28s
kube-prometheus-stack-operator-765dbf86b9-cqkpf          1/1     Running   0          2m28s
kube-prometheus-stack-prometheus-node-exporter-chbr9     1/1     Running   0          2m28s
kube-prometheus-stack-prometheus-node-exporter-zjd5c     1/1     Running   0          2m28s
prometheus-kube-prometheus-stack-prometheus-0            2/2     Running   0          2m19s
ubuntu@control-plane:~$
```

Retrieve the auto-generated Grafana admin password from the Kubernetes secret:

- kubectl get secret -n monitoring kube-prometheus-stack-grafana -o jsonpath="{.data.admin-password}" | base64 -d ; echo
- Default username: admin
- Password: Retrieved from the command above

```
ubuntu@control-plane:~$ kubectl get secret -n monitoring kube-prometheus-stack-grafana \
  -o jsonpath="{.data.admin-password}" | base64 -d ; echo
1eBeMuH6Srm07gFZ8Mtmj2jpZ0TzVwdnddxFbZv1
ubuntu@control-plane:~$
```

Grafana can then be accessed using **http://<Worker-Node-IP>:<Node-Port>**

6.8 Grafana Dashboard

1. Configure Prometheus as a Data Source

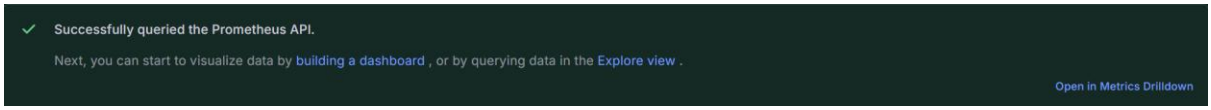
Access the Grafana UI and configure Prometheus as a data source. In most cases, kube-prometheus-stack automatically creates a Prometheus data source, but it is recommended to verify.

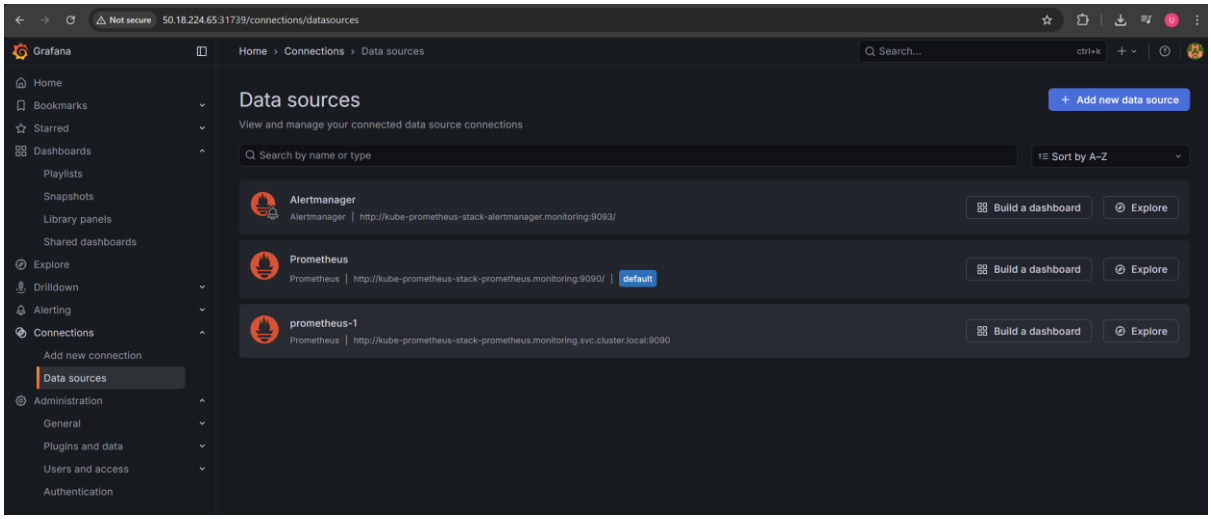
Steps in Grafana UI:

Navigate to **Connections** and select **Data Sources**. Click **Add data source** and select **Prometheus**. Set the URL to:

- http://kube-prometheus-stack-prometheus.monitoring.svc.cluster.local:9090

After adding the URL, click **Save & Test**. A successful message confirms that Grafana can query Prometheus metrics.



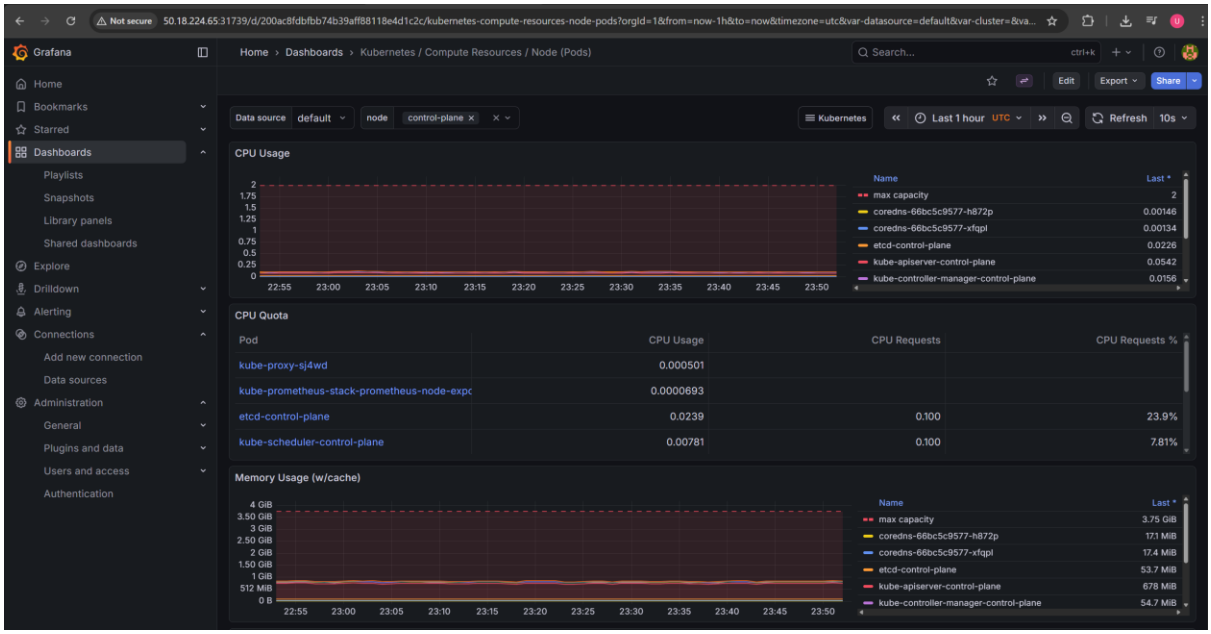


2. Pre-built Dashboards

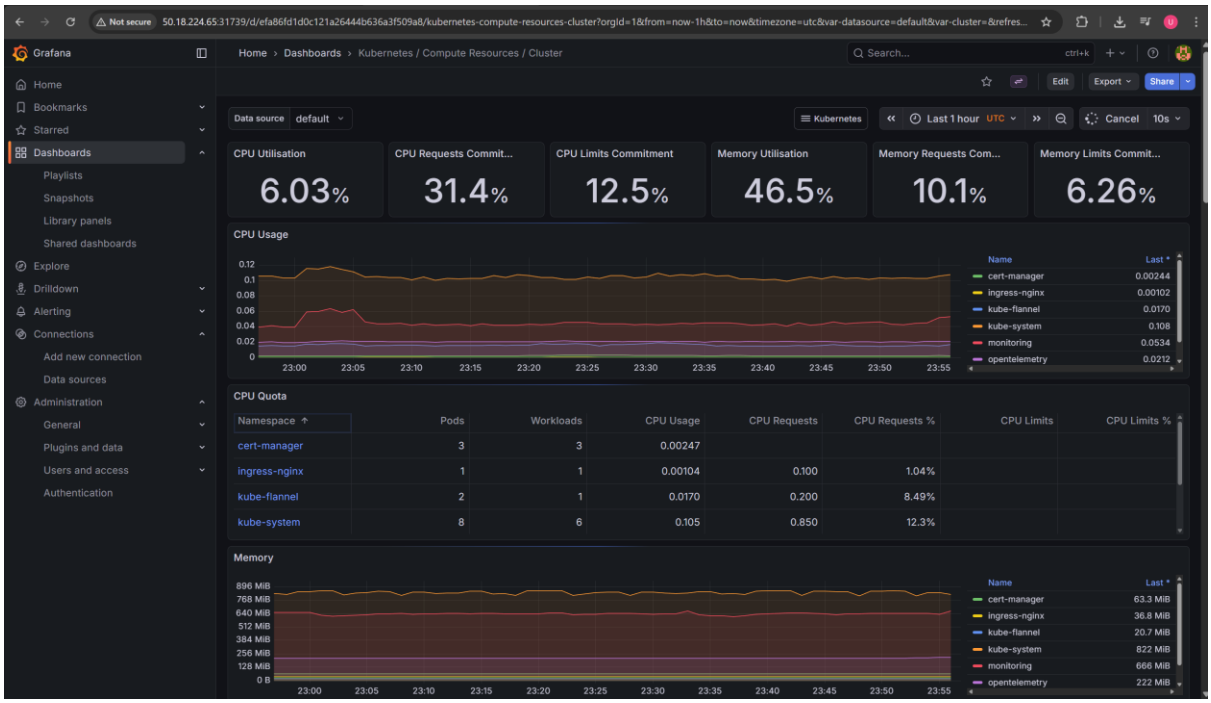
The kube-prometheus-stack includes several pre-configured dashboards for Kubernetes monitoring. These dashboards provide real-time visibility into cluster and workload performance.

Commonly used dashboards include:

- **Kubernetes / Compute Resources / Node (Pods)**
  Displays CPU and memory usage per node, along with pod-level resource consumption. This dashboard helps identify overloaded nodes or inefficient pod resource usage.



- **Kubernetes / Compute Resources / Cluster**
  Provides a cluster-wide view of resource utilization, namespace-level breakdowns, and overall capacity metrics.



These dashboards allow you to monitor cluster health, detect performance bottlenecks, and make informed scaling and optimization decisions.

7. Troubleshooting

The following issues were encountered during the Kubernetes cluster setup, networking configuration, application instrumentation, and observability stack deployment. Each issue describes the observed problem, underlying root cause, and the corrective action taken.

Issue 1: Calico system pods stuck in 0/1 state

Problem Description
Several pods in the calico-system namespace were not transitioning to a Running state and remained stuck at 0/1, even after initial configuration adjustments.

Root Cause
Calico was unable to bind to the correct network interface on the node. The primary network interface (ens5) was not explicitly configured in the Calico installation resource, causing partial initialization failures.

Solution

- Used ip addr show to identify the primary network interface (ens5)
- Edited the Calico installation resource using:
  - **kubectl edit installation default**
- Added the correct network interface configuration
- One of the affected pods transitioned to Running, while another continued to fail due to a separate issue (addressed in Issue 2)

Issue 2: Calico Typha connectivity failure (port 5473 timeout)

Problem Description
Calico logs showed repeated connection failures with the following error:

- **Failed to connect to typha endpoint 10.0.2.251:5473 ... error=dial tcp 10.0.2.251:5473: i/o timeout**

As a result, the control-plane node remained in a non-ready networking state.

Root Cause
Calico Typha, responsible for broadcasting network policy updates, was running on a worker node (10.0.2.251). The control-plane node (10.0.1.229) could not reach Typha because TCP port 5473 was blocked by the worker node's security group.

Without this connectivity, the control-plane could not receive the "ready" signal required to complete networking initialization.

Solution

- Updated the worker node security group to allow **Custom TCP traffic on port 5473**
- Once the port was opened, Typha communication succeeded and all Calico pods transitioned to Running

Issue 3: OpenTelemetry traces not appearing after Instrumentation deployment

Problem Description
After deploying the OpenTelemetry Instrumentation resource, no trace data appeared in the OpenTelemetry Collector logs or downstream systems.

Root Cause
The OTLP endpoint was configured using an incorrect Kubernetes service name:

- http://otel-collector.opentelemetry.svc:4317

This service name did not exist, preventing telemetry from reaching the collector.

Solution

Updated the endpoint to use the correct service DNS name:

- http://otel-collector-collector.opentelemetry.svc:4317

After applying the change and restarting the application pods, telemetry data began flowing successfully

Issue 4: OpenTelemetry Instrumentation not injected into application pods

Problem Description
Despite applying the Instrumentation resource, the Node.js application pods were not being instrumented, and no telemetry data was generated.

Root Cause
The Deployment manifest was missing the required annotation to enable automatic Node.js instrumentation injection. Without this annotation, the OpenTelemetry Operator did not modify the pod at startup.

Solution
Added the following annotation under the pod template metadata in the Deployment YAML:

- **instrumentation.opentelemetry.io/inject-nodejs: "opentelemetry/nodejs-instrumentation"**

After redeploying the application, the OpenTelemetry SDK was successfully injected at pod startup.

Issue 5: Application inaccessible despite Service and Ingress running

Problem Description
The Service and Ingress resources were created successfully, and their pods were running, but the application could not be accessed externally.

Root Cause
The Deployment was created in the opentelemetry namespace, while the Service and Ingress were created in the default namespace. Because Kubernetes services and ingresses are namespace-scoped, they could not route traffic to pods in a different namespace.

Solution

Updated the Service and Ingress manifest files to include **namespace: opentelemetry** and then re-applied the manifests. After aligning all resources in the same namespace, the application became accessible as expected