**Name space in the Kubernetes**

A **Namespace** in Kubernetes is a logical grouping mechanism that partitions a single Kubernetes cluster into multiple virtual clusters. Namespaces allow you to organize and isolate resources (like Pods, Services, Deployments) within the same physical cluster, making it easier to manage multi-tenant environments, teams, or projects.

It is just like an Azure Subnet in which similar/ related resources (like pods, services and deployments) of an application are placed in it.

**Example:** If you have an application which is build by two pods, three services and five deployments, these all resources are placed in a same/single namespace.

**Key points of Namespace:**

1. **Resource Isolation**:

* Namespaces help separate resources, making it easier to manage different projects, teams, or environments (e.g., dev, staging, production) within the same cluster.
* Prevent naming conflicts (e.g., two teams can have a web Deployment in separate namespaces).

1. **Logical Partitioning:**

* Namespaces provide a way to logically partition a single Kubernetes cluster into multiple virtual clusters.
* This allows you to organize resources and isolate them from each other.

1. **Access Control**

* You can set role-based access control (RBAC) permissions specific to each namespace.

1. **Organization:**

* Namespaces help organize resources within a cluster, making it easier to manage large and complex deployments.

1. **Environment Separation**

* Separate staging, production, and testing environments within the same cluster.

1. **Resource Quotas & Limits**

* Enforce CPU, memory, and storage limits per namespace.

1. **Multi-tenancy:**

* They enable multi-tenancy, allowing multiple teams or applications to share a single cluster without interfering with each other.

1. **Default Namespace**:

* If you don’t specify a namespace, Kubernetes uses the "default" namespace for your resources

In simple word Namespace is used to logically group all resources (pod, services, deployments) in order to manage the resource effectively.

### **Default Kubernetes Namespaces:**

Kubernetes comes with pre-defined namespaces:

| **Namespace** | **Purpose** |
| --- | --- |
| **default** | Used if no namespace is specified. |
| **kube-system** | Reserved for Kubernetes system components (e.g., kube-proxy, CoreDNS). |
| **kube-public** | Stores cluster-wide public resources (rarely used). |
| **kube-node-lease** | Tracks node health (internal use). |

**Block Diagram:**

Prod-env

DEV-env

P4

P3

P3

P2

P4

P2

P1

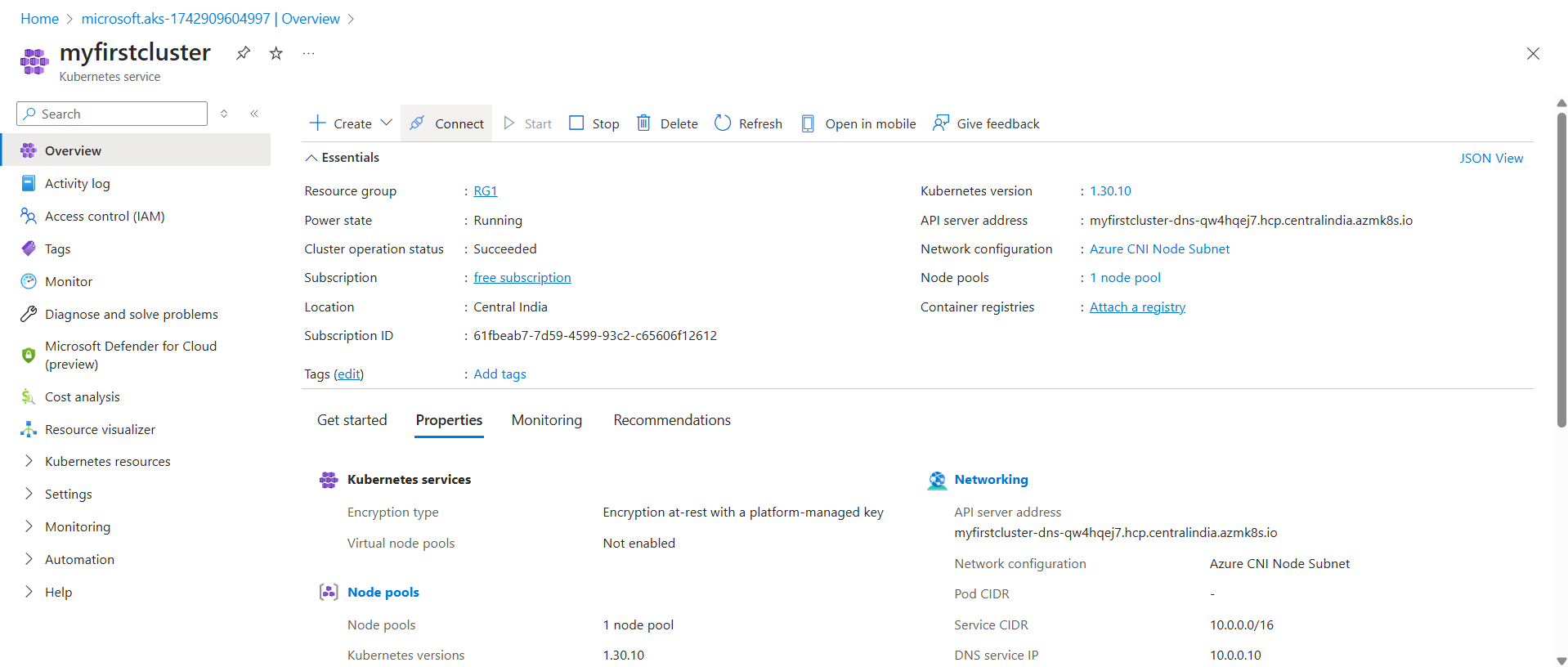
P1

Namespace-1

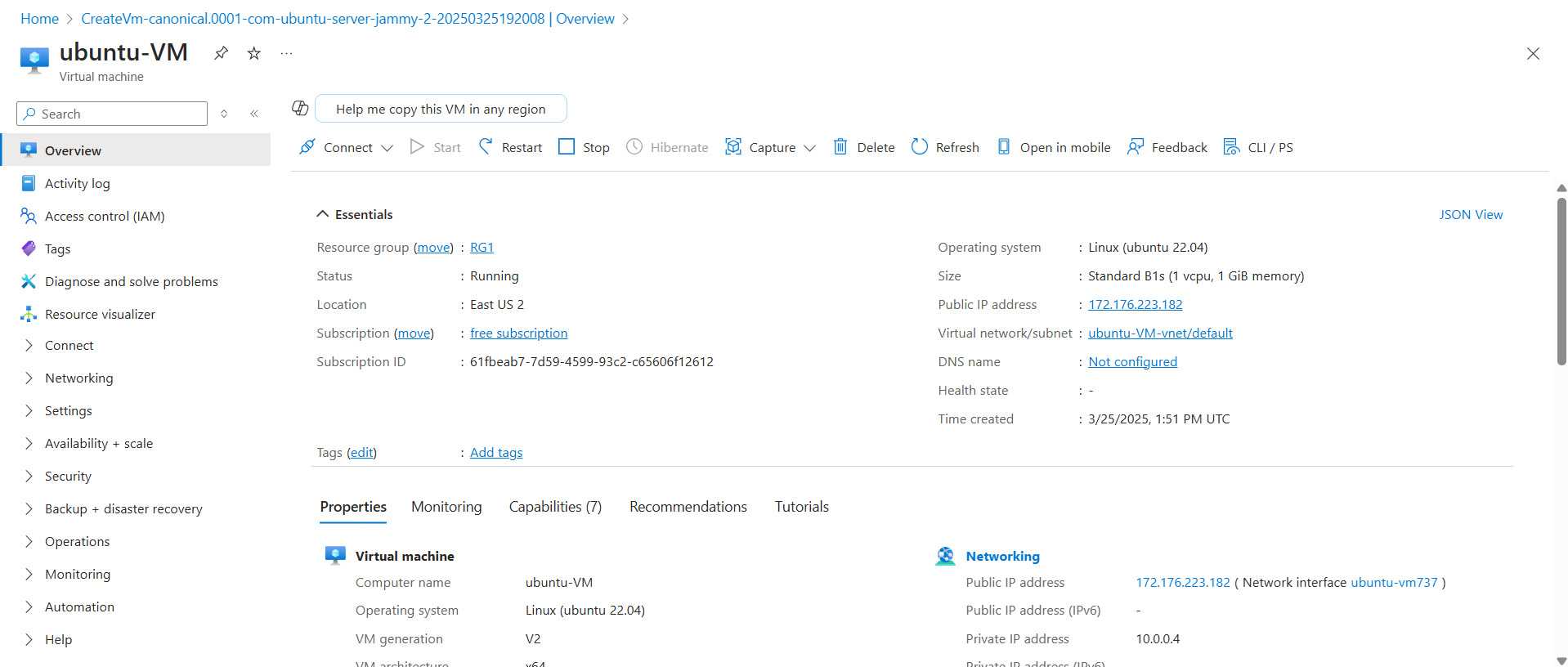
Namespace-2

Let’s work with Namespace practically:

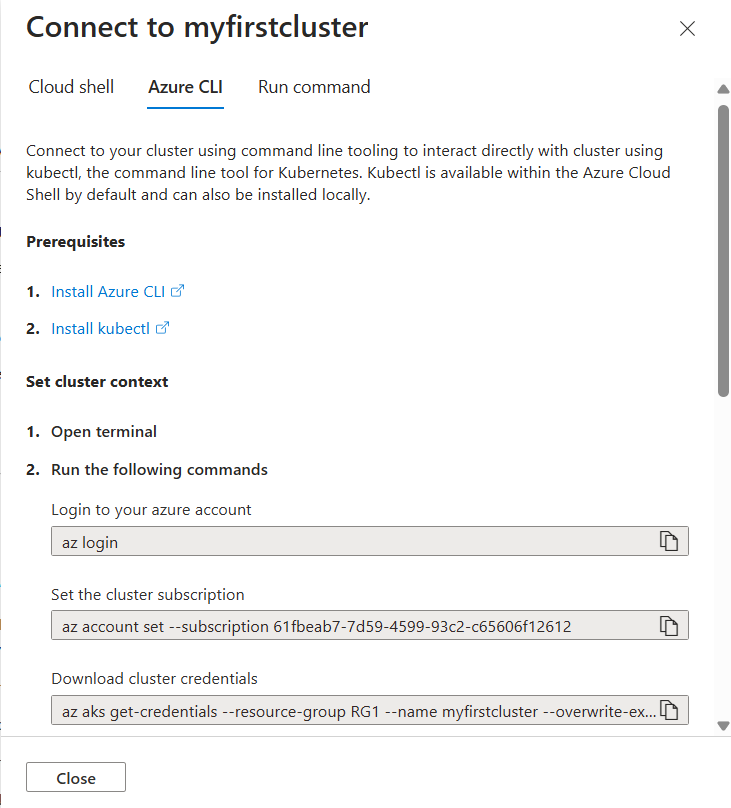
Step1: Create the Kubernetes cluster.



Step2: Create the Ubuntu VM.



**Step3:** Follow the below steps to connect the AKS to the Ubuntu VM.



**Fig:** Steps to follow in order to connect AKS using Azure CLI.

1. **URL to install Azure CLI:** curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
2. **Installing of Kubectl in Ubuntu VM:**
   1. Download the latest release with the command:

**Command:** curl -LO [https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl](https://dl.k8s.io/release/$(curl%20-L%20-s%20https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl)

* 1. Install Kubectl:

**Command:** sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl

#### Note: If you do not have root access on the target system, you can still install kubectl to the “~/.local/bin” directory:

chmod +x kubectl

mkdir -p ~/.local/bin

mv ./kubectl ~/.local/bin/kubectl

# and then append (or prepend) ~/.local/bin to $PATH

* 1. Test to ensure the version you installed is up-to-date:

**Command:** kubectl version --client

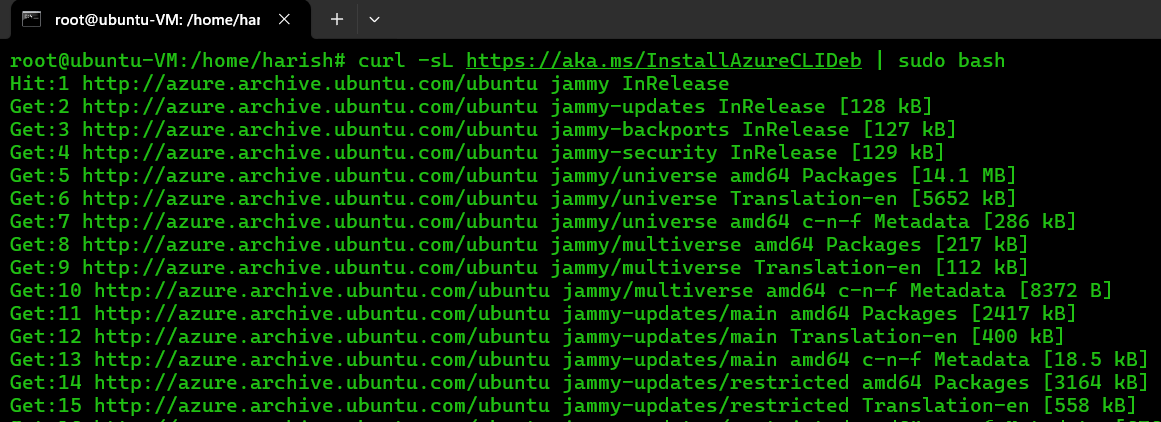


Fig: Azure CLI installed successfully.

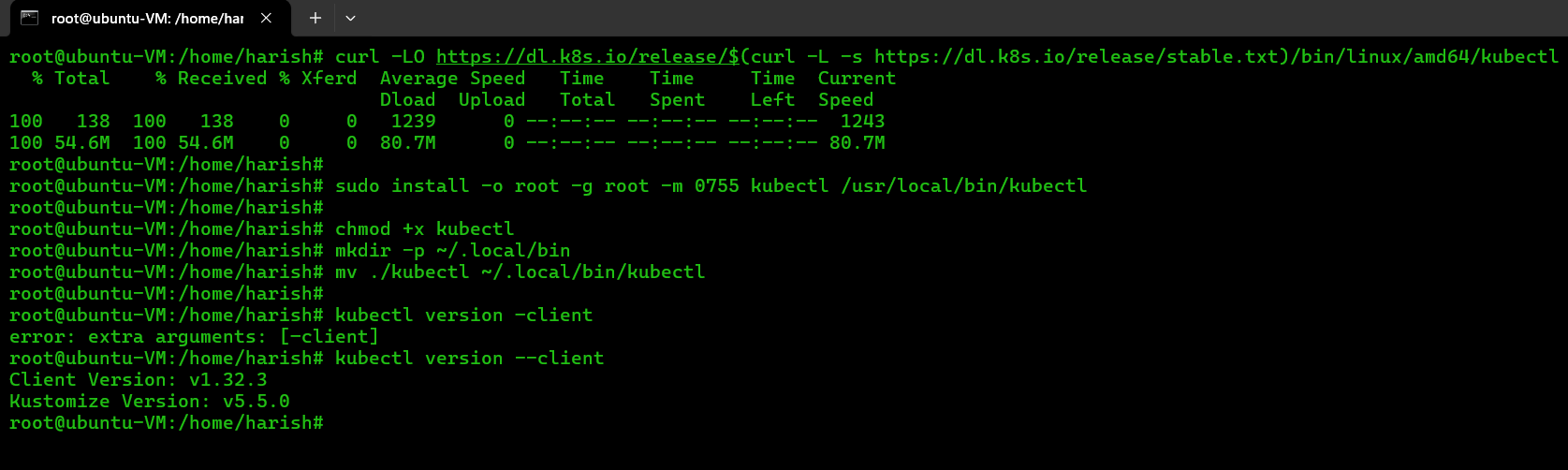


Fig: Kubectl is installed successfully.

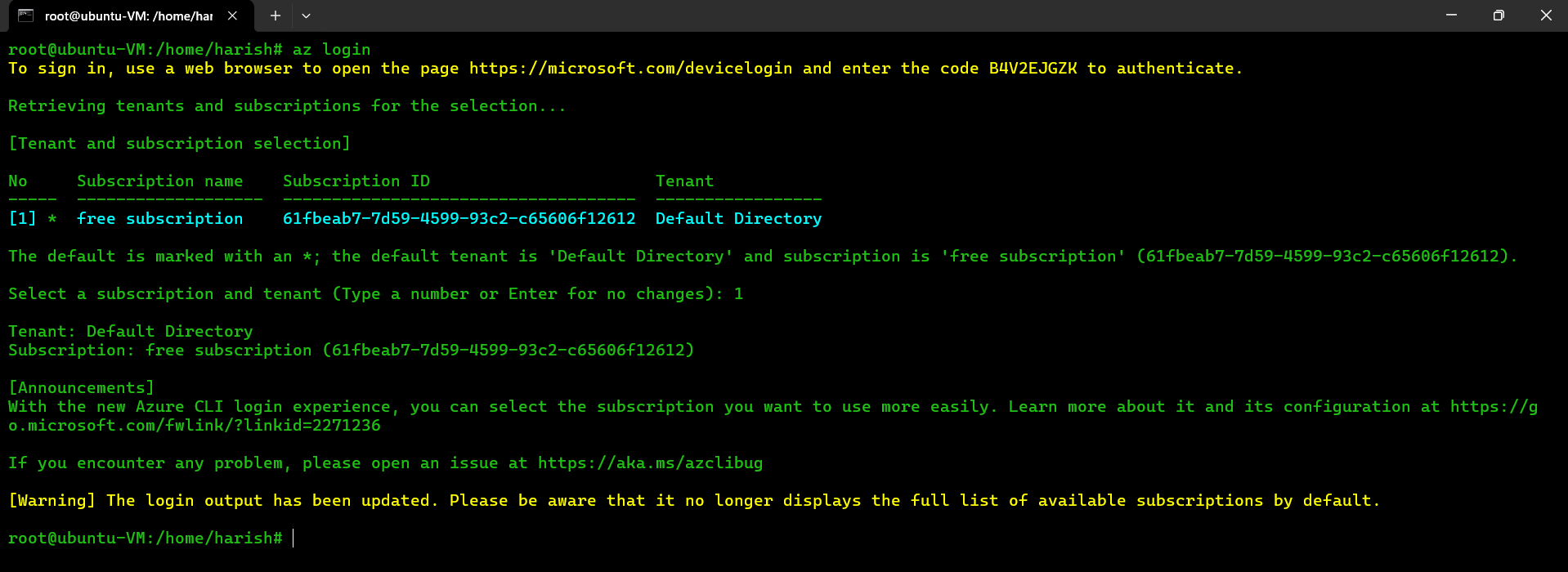


Fig: Login to the azure portal.

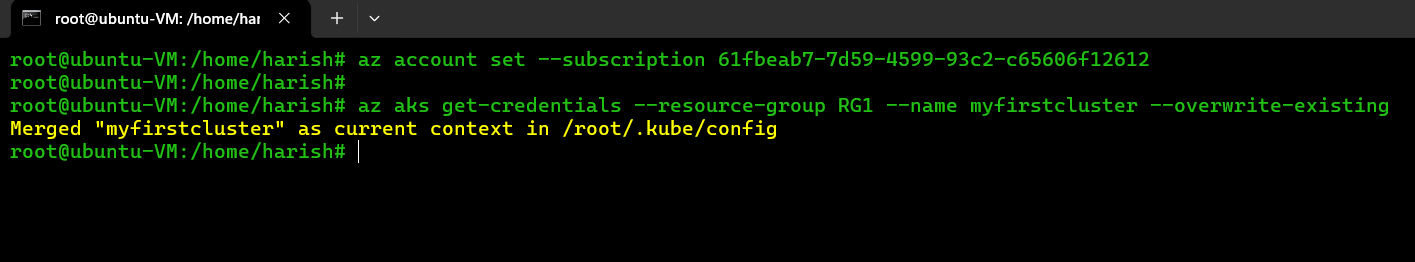
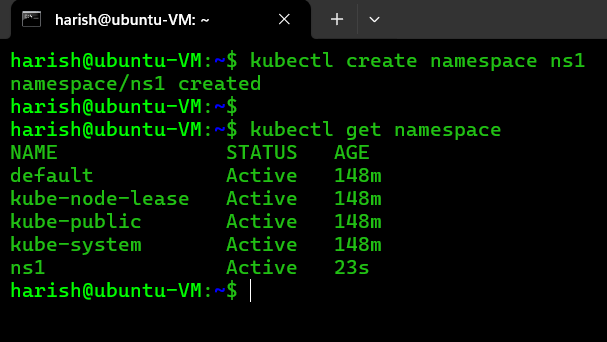


Fig: Setting of cluster subscription & Downloading of cluster credential.

Step4: Create a costume namespace (ns1).



Step4: Create a YAML file and deploy it to get a pod.

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-deployment

spec:

replicas: 2 # Number of pods to run

selector:

matchLabels:

app: my-app # Select pods with this label

template: # Pod template

metadata:

labels:

app: my-app # Labels for the pods

spec: # Pod specification

containers:

- name: my-container

image: nginx:latest # Container image

ports:

- containerPort: 80

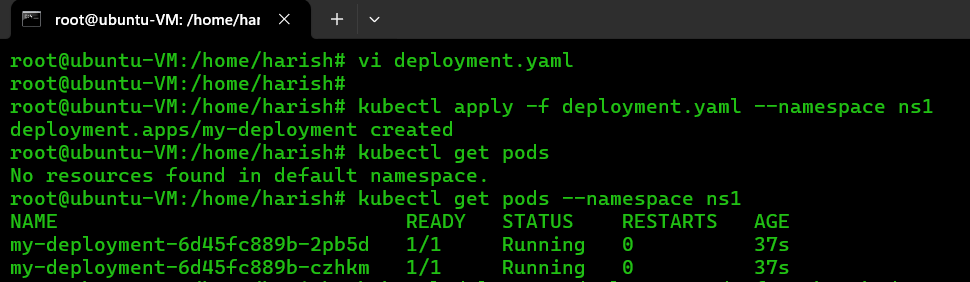
Fig: deployment.yaml file.

Fig: Two pods are created under namespace of ns1.

Step5: Let’s create one more namespace within this namespace deploy another pod.

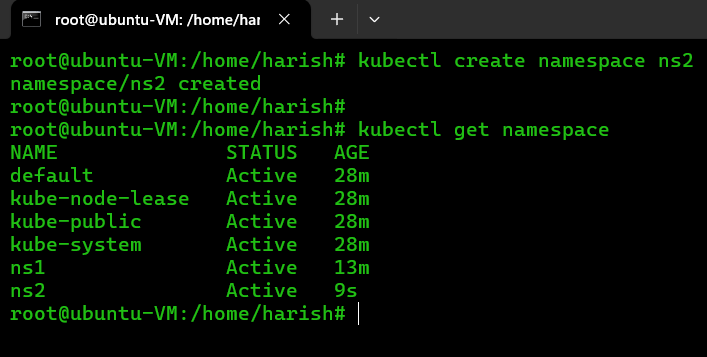


Fig: Namespace ns2.

Step6: Crete one more YAML file with same name and deploy it in ns2.

Fig: deploymet1.yaml file.

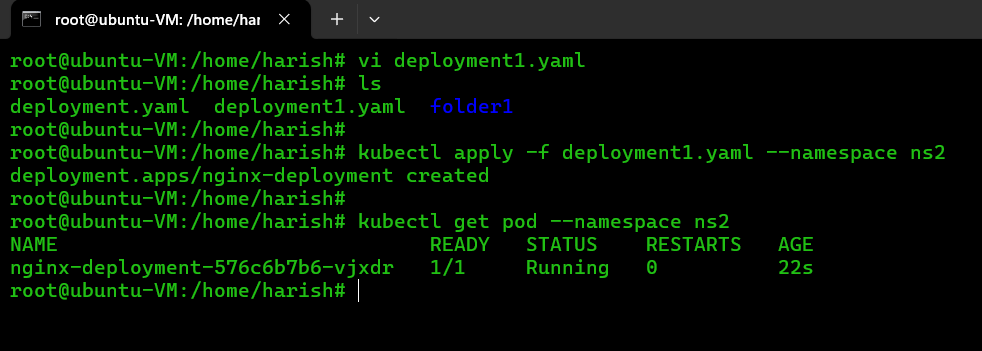


Fig: another pod is deployed in the new namespace ns2.

**Note:** Rather than viewing resources cluster-wide, you can scope your view to a specific namespace to list only relevant pods and services using below command

* 1. kubectl get pod --namespace <name-of-the-specific-namespace>
  2. kubectl get service --namespace <name-of-the-specific-namespace>

**Side Car/Helper/Supporting Containers:**

A **Sidecar Container** is a secondary container that runs alongside the main ("primary") container in the same Kubernetes **Pod**. It enhances or extends the functionality of the primary container without modifying its code.

Sidecars share the same network, storage, and lifecycle as the primary container, making them ideal for auxiliary tasks like logging, monitoring, or security.

**Key Features of Sidecar containers:**

1. **Separation of Concerns:**
   * Keeps the primary/main container focused on its core logic (e.g., running an app).
   * Offloads supporting tasks (e.g., logging, proxying) to the sidecar.
2. **They provide supporting services, such as:** 
   * **Logging:** Collecting and forwarding application logs.
   * **Monitoring:** Gathering and reporting application metrics.
   * **Proxies:** Handling network traffic or security concerns.
   * **Configuration:** managing configuration files.
3. **Shared Resources:**
   * Sidecars share the Pod’s network, storage, and lifecycle, simplifying communication.
4. **Lifecycle:**
   * They typically share the same lifecycle of the main application container, meaning they start and stop together with the Pod.

Let’s perform a task practically by using sidecar container.

Create three containers in a single pod, out of three one is main/application container in which Nginx is installed in it, and remaining two are sidecar containers. And then create a Volume and mount this volume to all these three containers.

We configured these two sidecar containers to print a message, such that sidecar container-1 prints **“Hi I am from Sidecar container”** and sidecar container-2 prints **“Hi I am from Sidecar container 2”.**

This messages will displayed while accessing the main container in the pod, because all this containers are mounted to a same volume in which these two messages are stored.

The creation of pod with three containers and mounting them to volume is done by deploying by configuring single YAML file.

**Block Diagram:**

Nginx (/var/www/html)

Sidecar-1

Main container

/var/log

Message-1

Sidecar-2

/var/log

Message-2

(/var/www/html)

Volume

Container engine

The blow given is the YAML file tow build three containers, one volume and mounting these contaniners to this volume:

apiVersion: apps/v1

kind: Deployment

metadata:

  creationTimestamp: null

  labels:

    app: nginx-webapp

  name: nginx-webapp

spec:

  replicas: 1

  selector:

    matchLabels:

      app: nginx-webapp

  strategy: {}

  template:

    metadata:

      creationTimestamp: null

      labels:

        app: nginx-webapp

    spec:

      containers:

      - image: busybox

        command: ["/bin/sh"]

        args: ["-c", "while true; do echo $(date -u) 'Hi I am from Sidecar container 1' >> /var/log/index.html; sleep 5; done"]

        name: sidecar-container1

        resources: {}

        volumeMounts:

        - name: var-logs

          mountPath: /var/log

      - image: busybox

        command: ["/bin/sh"]

        args: ["-c", "while true; do echo $(date -u) 'Hi I am from Sidecar container 2' >> /var/log/index.html; sleep 5; done"]

        name: sidecar-container2

        resources: {}

        volumeMounts:

        - name: var-logs

          mountPath: /var/log

      - image: nginx

        name: main-container

        resources: {}

        ports:

        - containerPort: 80

        volumeMounts:

        - name: var-logs

          mountPath: /usr/share/nginx/html

      dnsPolicy: Default

      volumes:

      - name: var-logs

        emptyDir: {}

status: {}

---

apiVersion: v1

kind: Service

metadata:

  name: nginx-webapp

  labels:

    run: nginx-webapp

spec:

  ports:

  - port: 80

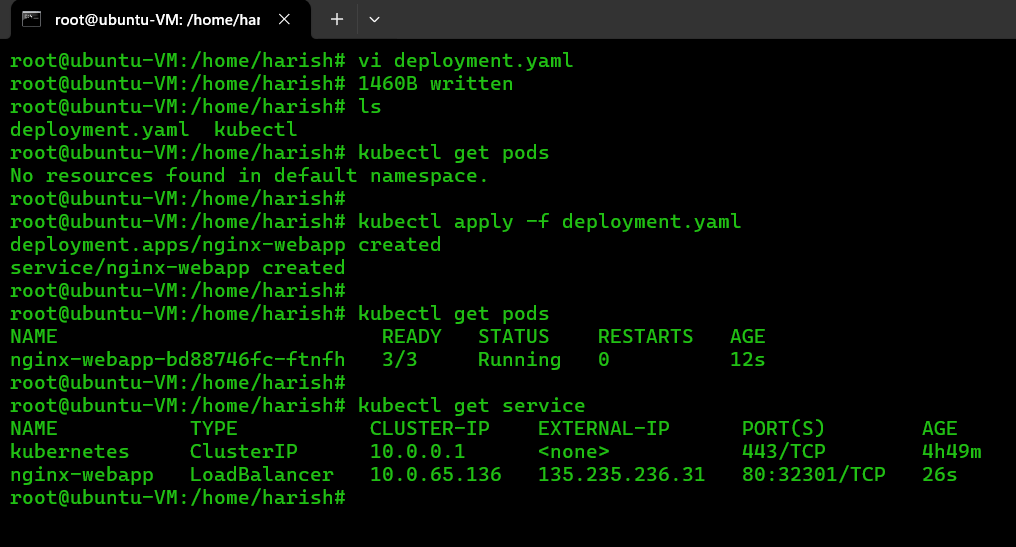
    protocol: TCP

  selector:

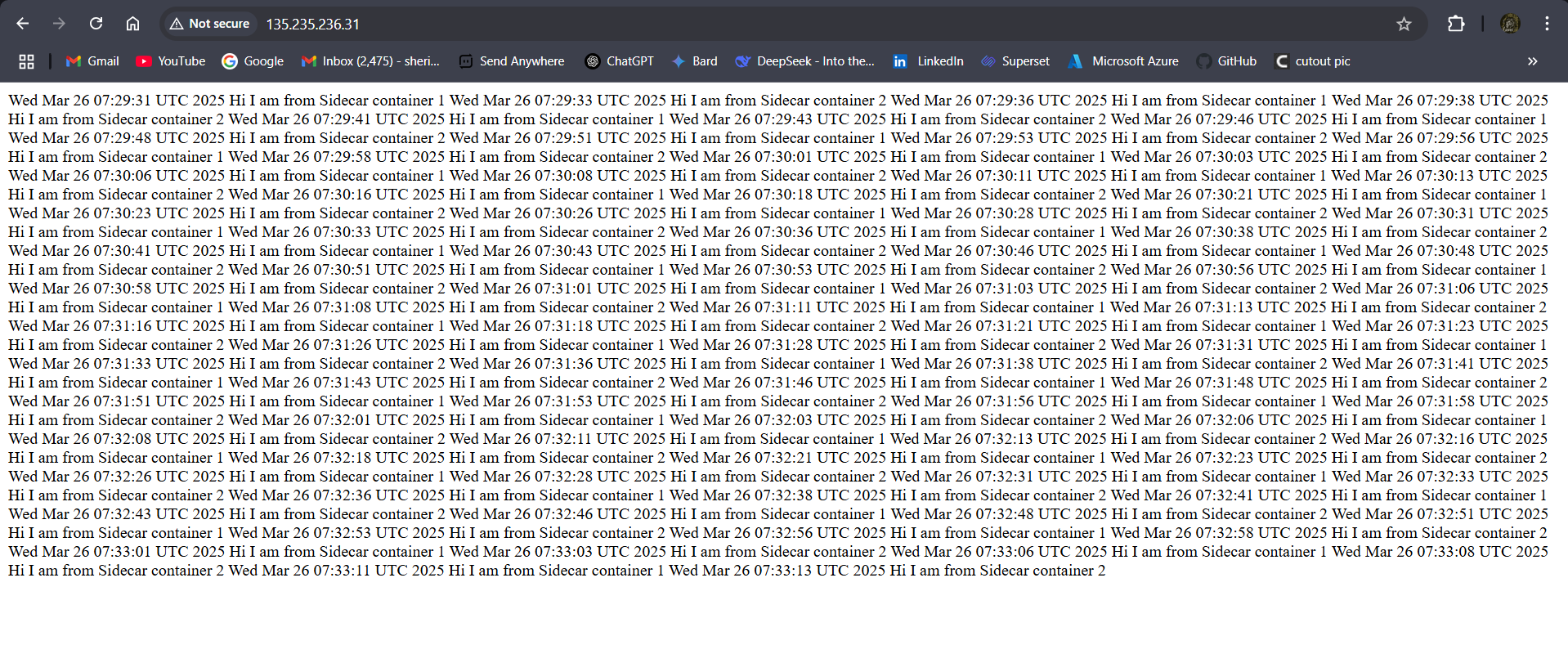
    app: nginx-webapp

  type: LoadBalancer

**Step1:** Deploy the above YAML file in the cluster by creating a deployment.yaml flile.



**Step2:** Let’s brows by using EXTERNAL-IP (135.235.236.31) of nginx-webapp service.



This all the messages are passed by the sidecar containers to the volume and this volume is mounted to the main container, through the main container we able so view this messages.

That means here the messages are generated by the sidecar container not the main container Nginx.

To avoid overloading the main application container (like Nginx) with message generation tasks, we use separate 'helper' containers called sidecars. These sidecars handle the message creation, keeping the main container running smoothly and efficiently."