Project title: GPU Clustering and Multi-Tenant Utilization Tool

Sponsor: SAIL Lab

PROFESSSOR: Dr.VAHID BEHZADAN (LinkedIn)
PROJECT ADVISOR: ASHISH AGARWAL(LinkedIn)

TEAM MEMBERS:
CHANDU GOGINENI(LinkedIn)
PAVAN TEJA SRIPATI (LinkedIn)
SNEHA VANGALA(LinkedIn)
KEERTHI SAI KASARANENI(LinkedIn)

Table of Contents

Brief summary of the project

- **Keywords**
  Key terms related to the project
- **Introduction**
  Background
  Problem Statement
  Objectives
  Scope
- **Literature Survey**
  Previous Work
  Gaps Identified
  Proposed Contribution
- **System Design and Methodology**
  System Architecture
  Methodology
  Tools and Technologies
- **Implementation**
  Detailed Implementation Steps
  Code Snippets
  Screenshots and Configuration
- **Results**
  Output Presentation (Tables, Charts, Screenshots)
  Analysis of Results
- **Discussion**
  Comparison with Expected Outcomes
  Challenges and Solutions
- **Conclusion**
  Summary of Achievements
  Contributions
- **Future Work**
- **References**

Abstract

The rise of GPU computing has revolutionized modern applications, enabling faster and more efficient execution of tasks such as machine learning, deep learning, and high-performance computing. However, the effective utilization of GPU resources in multi-user environments remains a challenge, particularly in areas such as resource allocation, monitoring, and ensuring user isolation. This project, **GPU Clustering and Multi-Tenant Utilization Tool**, addresses these challenges by developing a comprehensive solution for managing GPU resources across multiple users in a Kubernetes cluster.

The tool integrates GPU clustering, dynamic resource allocation, and an intuitive user interface inspired by platforms like JupyterHub and Google Colab, enabling seamless multi-tenant access. This report provides a step-by-step guide to deploying a high-performance, multi-tenant GPU environment using Kubernetes and JupyterHub. It details the setup of a robust Kubernetes cluster, the installation of essential tools, and the deployment of JupyterHub via Helm with advanced configurations. Key features include tailored resource profiles for GPU workloads, dynamic user scheduling, persistent storage, and the integration of the Kubernetes Dashboard for enhanced cluster management.

The project ensures efficient GPU resource utilization, scalability, and accessibility while addressing potential challenges with comprehensive troubleshooting guidelines. It is aimed at researchers, developers, and organizations requiring a scalable and equitable solution for GPU-intensive tasks. By simplifying GPU resource management and fostering collaboration in shared computing environments, this project significantly enhances productivity and resource efficiency.

Keywords

GPU Clustering, Kubernetes, Multi-Tenancy, JupyterHub, Resource Management, High-Performance Computing

Introduction:

The GPU Clustering and Multi-Tenant Utilization Tool project, sponsored by SAIL Lab, focuses on optimizing GPU resources in shared environments to address the growing demand for scalable and efficient GPU management. Modern applications in fields such as machine learning, deep learning, and scientific simulations rely heavily on Graphics Processing Units (GPUs) for their ability to process vast amounts of data rapidly. However, as organizations and institutions adopt GPUs for multi-user scenarios, challenges related to resource sharing, monitoring, and isolation arise.

Background

The increasing reliance on GPUs in multi-tenant environments introduces complexities such as under-utilization of resources, conflicts between users, and difficulty in providing equitable access. Traditional resource allocation mechanisms are often insufficient for GPU workloads, which demand dynamic and granular management. Without an effective solution, organizations face bottlenecks that hinder collaboration and productivity.

This project aims to bridge the gap by integrating GPU clustering, user management, and an interactive web-based interface into a unified solution. By leveraging Kubernetes, JupyterHub, and NVIDIA GPU monitoring tools, this tool empowers organizations to share GPU resources effectively while maintaining high levels of performance and user satisfaction.

Problem Statement

Organizations with shared GPU resources face the following key challenges:

Under-utilization and conflicts: Some GPUs remain idle while others are overburdened due to a lack of efficient scheduling and resource allocation.

Lack of user isolation: Multi-tenant environments require secure and isolated access to ensure one user's workload does not impact others.

Inefficient access methods: Users often lack intuitive interfaces to utilize shared GPU resources, making resource management cumbersome and error-prone.

These challenges create inefficiencies in GPU usage, limit scalability, and reduce overall productivity in shared computing environments.

What Does the Tool Do?

- **Share GPU Resources:** Users can access GPUs without worrying about conflicts, as the tool ensures fair allocation and scheduling.
- **Monitor Usage:** The tool shows who is using which GPU, how much memory is consumed, and whether the GPUs are idle or fully utilized.

- **Enable Collaboration:** Multiple users can run their tasks simultaneously on the same system while maintaining secure and isolated environments.

How Is It Built?

- Kubernetes: Manages the clustering of GPUs and containers for running user tasks.
- JupyterHub: Provides an interactive web-based platform for users to write code and run GPU tasks.
- NVIDIA GPU Plugins: Monitors and manages GPU resources within the Kubernetes cluster.
- Helm: Simplifies the deployment of JupyterHub on Kubernetes.

Why Is This Important?

In research labs, universities, or companies, GPUs are often shared among teams. Without proper management, these resources can become bottlenecks, slowing down work. This tool solves that problem by:

- Optimizing GPU usage.
- Allowing multiple users to work simultaneously without interference.
- Making it easier for users to access high-performance resources with a familiar interface.

This project is a practical solution for anyone looking to manage shared GPU resources effectively, whether in a small team or a larger organization. It ensures that everyone gets the power of GPUs when they need it, without wasting resources or causing conflicts.

Objectives

The project aims to develop a scalable, user-friendly tool for managing shared GPU resources in multi-tenant environments. The key objectives include:

- **GPU Clustering**: Combine multiple GPU systems into a unified resource pool using Kubernetes to enable dynamic allocation.
- **User Access Management**: Implement secure, role-based access control to prioritize resource allocation based on user needs.
- **Interactive Web Interface**: Provide an interface similar to Jupyter Notebooks or Google Colab for intuitive and seamless GPU access.

- **Resource Monitoring Dashboard**: Enable real-time monitoring of GPU usage, availability, and user activity through a comprehensive dashboard.
- **Scalability and Load Balancing**: Ensure efficient handling of simultaneous users with dynamic resource allocation and load balancing strategies.

Scope

The scope of this project includes:

- Designing and deploying a GPU clustering system using Kubernetes.
- Configuring and deploying JupyterHub as the primary user interface.
- Integrating NVIDIA GPU monitoring tools for real-time resource tracking.
- Implementing user authentication and access control mechanisms.
- Developing a resource management dashboard to monitor and optimize GPU usage.
- Testing scalability with multiple users and workloads.

This tool is designed for research labs, universities, and organizations that require efficient GPU resource sharing for multi-user environments.

Literature Survey

The management of GPU resources in multi-user environments has been a focus of numerous studies and solutions. Some key works include:

Kubernetes-based GPU Orchestration:

Kubernetes has become the de facto standard for container orchestration and is increasingly used to manage GPU workloads. NVIDIA's GPU Operator integrates GPU support into Kubernetes, enabling GPU-aware scheduling and resource allocation. However, it requires advanced expertise to configure and manage in multi-tenant setups.

JupyterHub for Collaborative Computing:

JupyterHub has been widely adopted for providing a collaborative, multi-user environment. It allows users to run notebooks in isolated containers but lacks native support for GPU resource management and monitoring in a clustered environment.

CUDA Multi-Process Service (MPS):

NVIDIA's CUDA MPS improves the sharing of GPUs among multiple processes by reducing context switching overhead. However, MPS is limited to specific types of workloads and does not support robust user isolation or role-based access control.

Resource Management Frameworks:

- **Mesos**: A resource management platform for distributed systems that supports GPU resource allocation. However, its adoption has declined due to the growing popularity of Kubernetes.
- **Slurm**: A workload manager designed for high-performance computing clusters. While Slurm provides GPU support, it lacks the flexibility and ease of use provided by Kubernetes and JupyterHub integrations.

Gaps Identified

Despite advancements in GPU resource management, several gaps remain unaddressed:

- **Limited Multi-Tenant Support**: Existing solutions lack robust multi-tenant capabilities, making it difficult to ensure fair resource sharing, isolation, and security in shared environments.
- **User-Friendly Interfaces**: Tools like Kubernetes and Slurm are powerful but require significant technical expertise, which hinders accessibility for non-technical users such as researchers or students.
- **Integration Challenges**: While standalone solutions like JupyterHub and Kubernetes offer distinct capabilities, integrating them for GPU-specific tasks is complex and lacks comprehensive documentation.
- **Dynamic Resource Allocation**: Most solutions fail to dynamically allocate resources based on real-time demand and workload priorities, leading to inefficient GPU utilization.

- **Scalability**: Current implementations often struggle to handle the demands of large-scale, multi-user environments without significant manual configuration and monitoring.

Proposed Contribution

This project addresses the identified gaps by proposing the **GPU Clustering and Multi-Tenant Utilization Tool**, which integrates Kubernetes, JupyterHub, and GPU resource monitoring into a unified solution. The key contributions include:

Comprehensive Multi-Tenant GPU Management:

- Enables secure, role-based access control and resource quotas for multiple users.
- Provides user isolation to prevent interference between tasks.

Dynamic GPU Resource Allocation:

- Implements intelligent scheduling to maximize GPU utilization across users and workloads.
- Balances workloads dynamically to adapt to varying demands.

User-Friendly Web Interface:

- Offers an intuitive interface inspired by Jupyter Notebooks and Google Colab, reducing the learning curve for users.
- Displays real-time GPU usage metrics, simplifying resource monitoring.

Integration of Key Technologies:

- Combines Kubernetes' powerful orchestration capabilities with JupyterHub's interactive computing environment.
- Utilizes NVIDIA's GPU Operator and Helm charts for seamless deployment and management.

Scalability and Accessibility:

- Ensures that the system can handle simultaneous requests from multiple users without compromising performance.

- Tailors resource profiles for diverse workloads, from lightweight tasks to GPU-intensive computations.

Comprehensive Documentation:

- Provides detailed deployment and troubleshooting guidelines to simplify adoption for research labs, universities, and organizations.

By addressing these gaps, the project enables organizations to maximize the potential of their GPU resources, foster collaboration, and ensure efficient resource utilization in multi-tenant environments.

System Architecture

The architecture consists of the following layers:

Infrastructure Layer:

- Physical or virtual GPU-enabled servers.
- Operating systems with NVIDIA GPU drivers and Kubernetes prerequisites.
- Kubernetes cluster as the backbone for orchestration.

Resource Management Layer:

- Kubernetes manages pods, nodes, and GPU workloads.
- NVIDIA GPU Operator facilitates GPU scheduling and monitoring.
- Helm automates the deployment of applications like JupyterHub.

User Interface Layer:

- JupyterHub provides an interactive, web-based notebook environment.
- Flask-based web UI offers simplified resource management and user access control.

Monitoring and Analytics Layer:

- Kubernetes Dashboard for visualizing cluster status.

- Custom dashboards for real-time GPU utilization and user activity monitoring.

Components Overview

Kubernetes:

- Orchestrates GPU workloads across the cluster.
- Ensures scalability, load balancing, and fault tolerance.
- Supports NVIDIA GPU plugins for GPU-aware scheduling.

NVIDIA GPU Operator:

- Manages GPU resources in the Kubernetes cluster.
- Enables device plugins for exposing GPUs to pods.
- Provides tools for monitoring GPU health and usage.

JupyterHub:

- Acts as the primary user interface for accessing GPU resources.
- Supports multi-user environments with isolated user sessions.
- Allows users to run code interactively while leveraging assigned GPU resources.

Helm:

- Simplifies the deployment of complex applications, including JupyterHub.
- Provides reusable templates and version control for Kubernetes deployments.

Flask:

- Builds a custom web interface for resource management.
- Handles user authentication, token management, and API integration with Kubernetes.

Kubernetes Dashboard:

- Visualizes node health, pod status, and resource utilization.
- Provides administrative controls for cluster management.

Storage Backend:

- Ensures persistent storage for user notebooks and data.
- Configured using Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) in Kubernetes.

Methodology

The system is developed using a step-by-step approach to ensure modularity and scalability.

System Initialization:

- Install and configure Kubernetes on GPU-enabled servers.
- Set up NVIDIA GPU drivers and container runtimes (Docker or containerd).
- Deploy the NVIDIA GPU Operator for GPU resource management.

Cluster Setup:

- Use kubeadm to initialize the Kubernetes cluster.
- Apply a pod network add-on (e.g., Flannel) to enable communication between nodes.
- Verify node readiness and cluster health.

Application Deployment:

- Deploy JupyterHub using Helm charts with GPU-specific configurations.
- Configure user roles, access levels, and resource profiles for multi-tenant usage.
- Integrate Flask for custom resource management and user authentication.

Monitoring and Troubleshooting:

- Set up the Kubernetes Dashboard for real-time cluster monitoring.
- Use GPU-specific monitoring tools to track GPU utilization and health.
- Include logging mechanisms for debugging and performance analysis.

Testing and Optimization:

- Conduct load testing to ensure the system handles multiple users and workloads.
- Optimize GPU scheduling and resource allocation strategies.
- Address user feedback to enhance the interface and functionality.

Tools and Technologies

Kubernetes

- **Role**: Core orchestration platform.
- **Key Features**: Scalability, fault tolerance, and resource management.
- **Why**: Industry-standard for container orchestration and workload management.

Docker/Containerd

- **Role**: Container runtimes for running applications in isolated environments.
- **Key Features**: Portability, efficiency, and compatibility with Kubernetes.
- **Why**: Ensures reproducibility and simplifies application deployment.

NVIDIA GPU Operator

- **Role**: Facilitates GPU integration with Kubernetes.
- **Key Features**: Device plugins, GPU health monitoring, and advanced scheduling.
- **Why**: Provides seamless support for GPU workloads in Kubernetes.

Helm

- **Role**: Kubernetes package manager for deploying and managing applications.
- **Key Features**: Reusable templates, version control, and ease of deployment.
- **Why**: Simplifies the deployment of JupyterHub and other complex applications.

JupyterHub

- **Role**: Provides a collaborative, web-based notebook interface for users.
- **Key Features**: Multi-user support, session isolation, and integration with Kubernetes.
- **Why**: Familiar and user-friendly interface for researchers and developers.

Flask

- **Role**: Builds a custom web-based UI for user and resource management.
- **Key Features**: Lightweight, extensible, and easy to integrate with APIs.
- **Why**: Provides a simple interface for managing user access and viewing resource usage.

Kubernetes Dashboard

- **Role**: Visualizes cluster status, node health, and resource usage.
- **Key Features**: Web-based interface, real-time monitoring, and administrative controls.
- **Why**: Helps administrators monitor and manage the cluster effectively.

NVIDIA GPU Plugins

- **Role**: Enables Kubernetes to schedule GPU workloads and track GPU metrics.
- **Key Features**: Exposes GPUs to containers, supports device health checks.
- **Why**: Integrates GPU capabilities directly into the Kubernetes environment.

Flannel

- **Role**: Provides pod networking for Kubernetes.
- **Key Features**: Simple, efficient, and widely compatible.
- **Why**: Ensures seamless communication between pods in the cluster.

Persistent Storage

- **Role**: Ensures data persistence for user notebooks and applications.

- **Key Features**: PVs, PVCs, and dynamic storage provisioning.
- **Why**: Enables users to retain their data across sessions and reboots.

Implementation Details

Overview

The goal of this setup is to deploy Kubernetes in a containerized environment to efficiently manage workloads, monitor resources in real-time, and allocate resources dynamically. This setup includes the integration of JupyterHub on Kubernetes, enabling users to execute interactive notebooks with optimized resource management.

Prerequisites

- Operating System: Ubuntu 20.04 (or higher)
- Permissions: Sudo or root access for installation and configuration
- Hardware: Adequate system resources for running Kubernetes components (e.g., sufficient CPU, RAM, and storage)

Docker Installation

Docker is a container runtime used to run Kubernetes pods. Below are the steps to install and configure Docker:

Update the Package Index:

```
sudo apt-get update
```

Install Dependencies for HTTPS Repository:

```
sudo apt-get install -y \
  apt-transport-https \

  ca-certificates \

  curl \

  gnupg \

  lsb-release
```

Add Docker's Official GPG Key:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-
keyring.gpg
```

Add Docker Repository:

```
echo \

 "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-
archive-keyring.gpg] https://download.docker.com/linux/ubuntu \

 $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Install Docker Engine:

```
sudo apt-get update

sudo apt-get install -y docker-ce docker-ce-cli containerd.io
```

Verify Installation:

```
sudo docker run hello-world
```

Kubernetes Installation

Download the latest release:

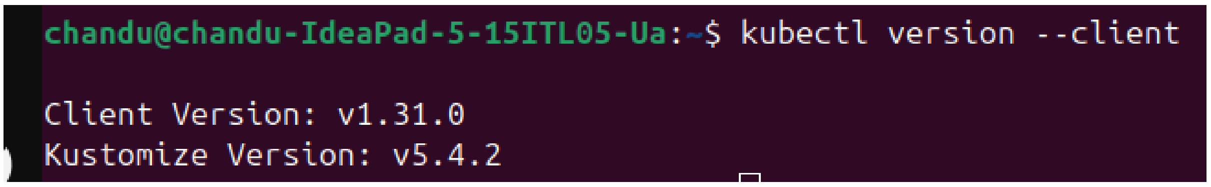
```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
```

Kubectl Installation:

```
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Verify Installation:

```
kubectl version --client
```



```
chandu@chandu-IdeaPad-5-15ITL05-Ua:~$ kubectl version --client

Client Version: v1.31.0
Kustomize Version: v5.4.2
```

kubeadm Installation

Install Dependencies:

```
sudo apt-get update

sudo apt-get install -y apt-transport-https ca-certificates curl
```

Download the Google Cloud public signing key:

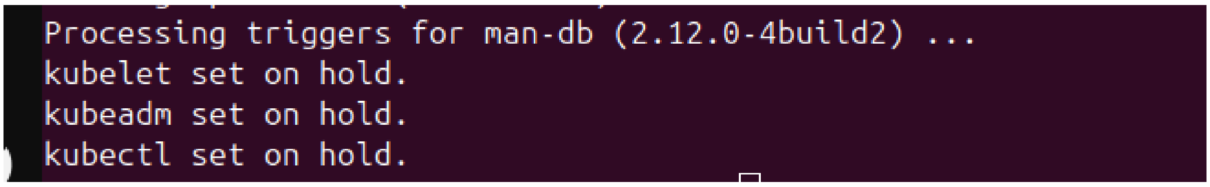
```
sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg
https://packages.cloud.google.com/apt/doc/apt-key.gpg
```

Add Kubernetes Repository:

```
echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc/apt/sources.list.d/kubernetes.list
```

Install kubelet, kubeadm, and kubectl:

```
sudo apt-get update

sudo apt-get install -y kubelet kubeadm kubectl

sudo apt-mark hold kubelet kubeadm kubectl
```



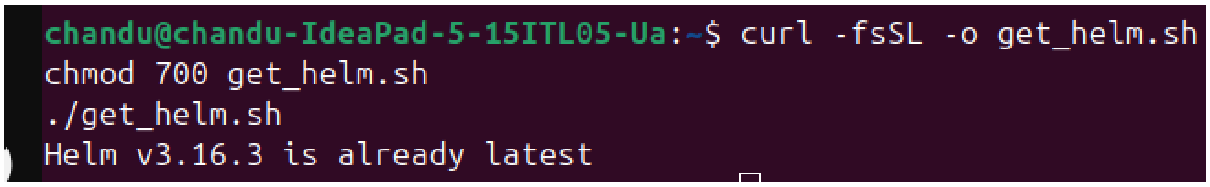
```
Processing triggers for man-db (2.12.0-4build2) ...
kubelet set on hold.
kubeadm set on hold.
kubectl set on hold.
```

Helm Installation script download:

```
curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
```

Make the script executable and run the script:

```
chmod 700 get_helm.sh
./get_helm.sh
```



```
chandu@chandu-IdeaPad-5-15ITL05-Ua:~$ curl -fsSL -o get_helm.sh
chmod 700 get_helm.sh
./get_helm.sh
Helm v3.16.3 is already latest
```

Verify Installation:

```
helm version
```

Kubernetes Cluster Setup

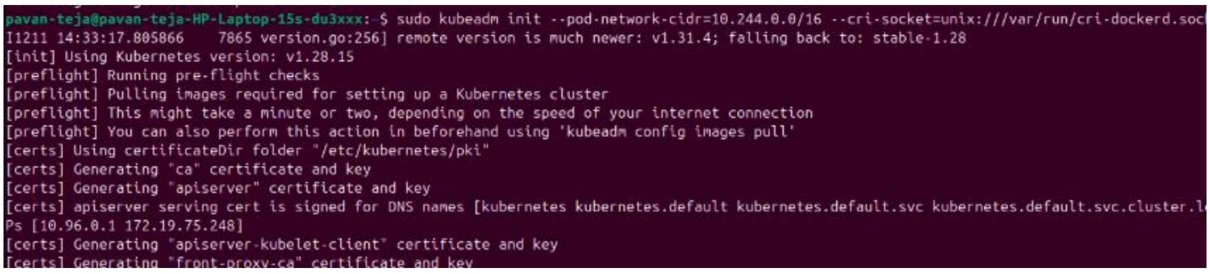
Disable Swap: *Kubernetes requires swap to be disabled for optimal functioning*

```
sudo swapoff -a
```

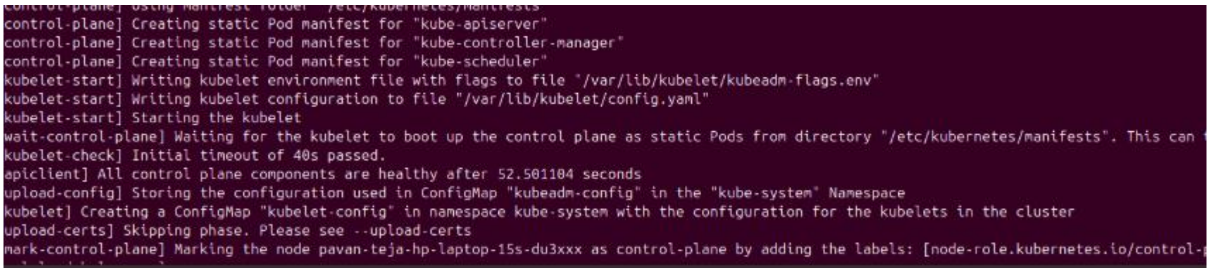
Initialize Kubernetes Cluster

Run the following command to initialize the master node:

```
sudo kubeadm init --pod-network-cidr=10.244.0.0/16 --cri-socket=unix:///var/run/cri-dockerd.sock
```



```
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$ sudo kubeadm init --pod-network-cidr=10.244.0.0/16 --cri-socket=unix:///var/run/cri-dockerd.soc
I1211 14:33:17.805866    7865 version.go:256] remote version is much newer: v1.31.4; falling back to: stable-1.28
[init] Using Kubernetes version: v1.28.15
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [kubernetes kubernetes.default kubernetes.default.svc kubernetes.default.svc.cluster.l
Ps [10.96.0.1 172.19.75.248]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
```



```
control-plane] Creating static Pod manifest for "kube-apiserver"
control-plane] Creating static Pod manifest for "kube-controller-manager"
control-plane] Creating static Pod manifest for "kube-scheduler"
kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
kubelet-start] Starting the kubelet
wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods from directory "/etc/kubernetes/manifests". This can
kubelet-check] Initial timeout of 40s passed.
apiclient] All control plane components are healthy after 52.501104 seconds
upload-config] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-system" Namespace
kubelet] Creating a ConfigMap "kubelet-config" in namespace kube-system with the configuration for the kubelets in the cluster
upload-certs] Skipping phase. Please see --upload-certs
mark-control-plane] Marking the node pavan-teja-hp-laptop-15s-du3xxx as control-plane by adding the labels: [node-role.kubernetes.io/control-
```

- The command outputs a token to join worker nodes to the cluster.

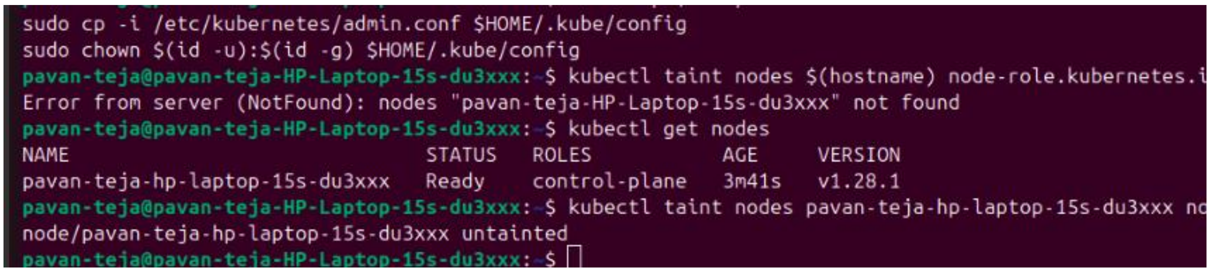
Configure kubectl for User Access

```
mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Untaint the Control Plane Node

Allow scheduling of pods on the control-plane node:

```
kubectl taint nodes $(hostname) node-
role.kubernetes.io/control-plane-
```



```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$ kubectl taint nodes $(hostname) node-role.kubernetes.i
Error from server (NotFound): nodes "pavan-teja-HP-Laptop-15s-du3xxx" not found
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$ kubectl get nodes
NAME                              STATUS   ROLES           AGE     VERSION
pavan-teja-hp-laptop-15s-du3xxx   Ready    control-plane   3m41s   v1.28.1
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$ kubectl taint nodes pavan-teja-hp-laptop-15s-du3xxx no
node/pavan-teja-hp-laptop-15s-du3xxx untainted
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$
```

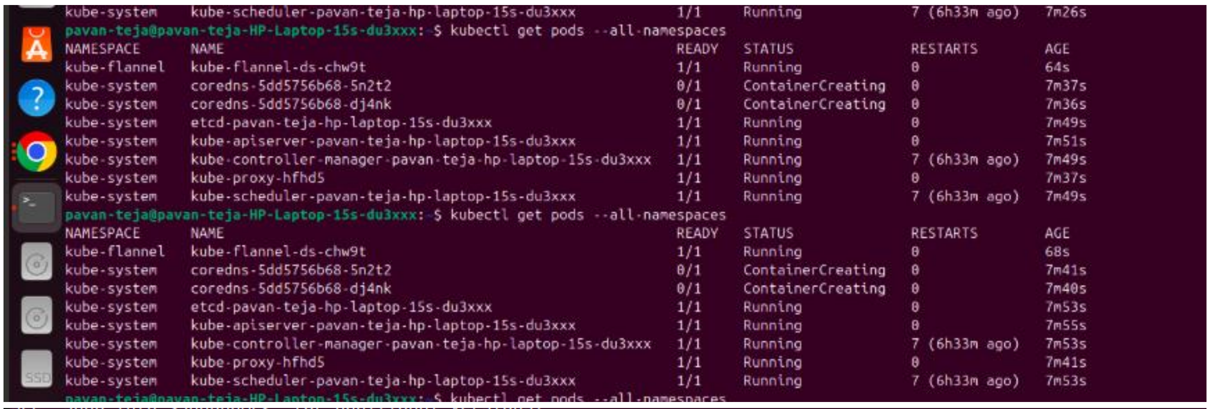
Networking Setup

Install the Flannel network plugin for inter-pod communication:

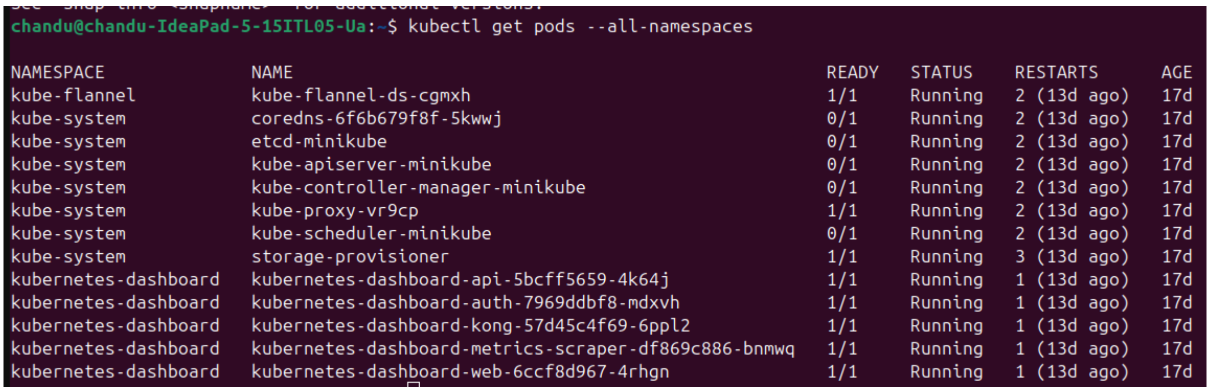
```
kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube
-flannel.yml-flannel.yml
```

<u>Verify all pods are running:</u>

```
kubectl get pods --all-namespaces
```



```
kube-system    kube-scheduler-pavan-teja-hp-laptop-15s-du3xxx          1/1     Running              7 (6h33m ago)   7m26s
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$ kubectl get pods --all-namespaces
NAMESPACE      NAME                                                    READY   STATUS               RESTARTS        AGE
kube-flannel   kube-flannel-ds-chw9t                                   1/1     Running              0               64s
kube-system    coredns-5dd5756b68-5n2t2                                0/1     ContainerCreating    0               7m37s
kube-system    coredns-5dd5756b68-dj4nk                                0/1     ContainerCreating    0               7m36s
kube-system    etcd-pavan-teja-hp-laptop-15s-du3xxx                    1/1     Running              0               7m49s
kube-system    kube-apiserver-pavan-teja-hp-laptop-15s-du3xxx          1/1     Running              0               7m51s
kube-system    kube-controller-manager-pavan-teja-hp-laptop-15s-du3xxx 1/1     Running              7 (6h33m ago)   7m49s
kube-system    kube-proxy-hfhd5                                        1/1     Running              0               7m37s
kube-system    kube-scheduler-pavan-teja-hp-laptop-15s-du3xxx          1/1     Running              7 (6h33m ago)   7m49s
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$ kubectl get pods --all-namespaces
NAMESPACE      NAME                                                    READY   STATUS               RESTARTS        AGE
kube-flannel   kube-flannel-ds-chw9t                                   1/1     Running              0               68s
kube-system    coredns-5dd5756b68-5n2t2                                0/1     ContainerCreating    0               7m41s
kube-system    coredns-5dd5756b68-dj4nk                                0/1     ContainerCreating    0               7m40s
kube-system    etcd-pavan-teja-hp-laptop-15s-du3xxx                    1/1     Running              0               7m53s
kube-system    kube-apiserver-pavan-teja-hp-laptop-15s-du3xxx          1/1     Running              0               7m55s
kube-system    kube-controller-manager-pavan-teja-hp-laptop-15s-du3xxx 1/1     Running              7 (6h33m ago)   7m53s
kube-system    kube-proxy-hfhd5                                        1/1     Running              0               7m41s
kube-system    kube-scheduler-pavan-teja-hp-laptop-15s-du3xxx          1/1     Running              7 (6h33m ago)   7m53s
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$ kubectl get pods --all-namespaces
```

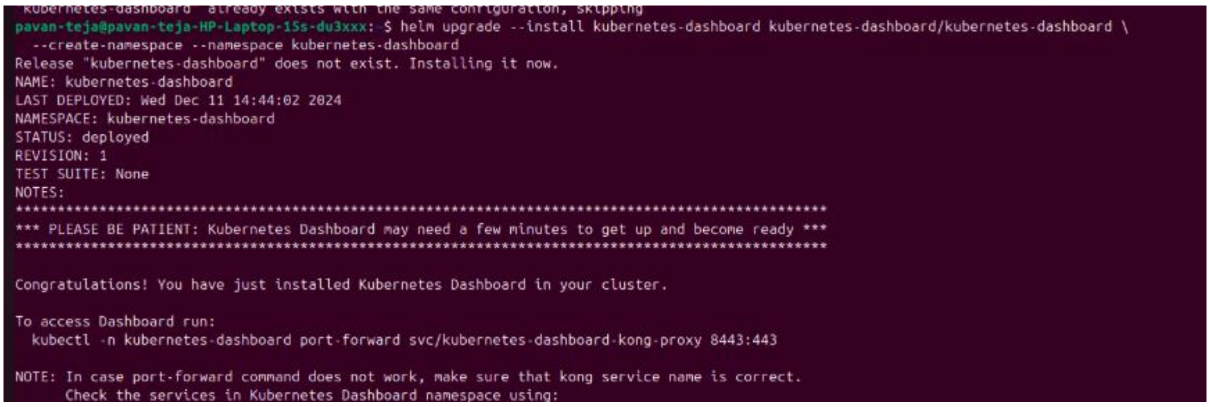


```
chandu@chandu-IdeaPad-5-15ITL05-Ua:~$ kubectl get pods --all-namespaces

NAMESPACE              NAME                                                  READY   STATUS    RESTARTS      AGE
kube-flannel           kube-flannel-ds-cgmxh                                 1/1     Running   2 (13d ago)   17d
kube-system            coredns-6f6b679f8f-5kwwj                              0/1     Running   2 (13d ago)   17d
kube-system            etcd-minikube                                         0/1     Running   2 (13d ago)   17d
kube-system            kube-apiserver-minikube                               0/1     Running   2 (13d ago)   17d
kube-system            kube-controller-manager-minikube                      0/1     Running   2 (13d ago)   17d
kube-system            kube-proxy-vr9cp                                      1/1     Running   2 (13d ago)   17d
kube-system            kube-scheduler-minikube                               0/1     Running   2 (13d ago)   17d
kube-system            storage-provisioner                                   1/1     Running   3 (13d ago)   17d
kubernetes-dashboard   kubernetes-dashboard-api-5bcff5659-4k64j              1/1     Running   1 (13d ago)   17d
kubernetes-dashboard   kubernetes-dashboard-auth-7969ddbf8-mdxvh             1/1     Running   1 (13d ago)   17d
kubernetes-dashboard   kubernetes-dashboard-kong-57d45c4f69-6ppl2            1/1     Running   1 (13d ago)   17d
kubernetes-dashboard   kubernetes-dashboard-metrics-scraper-df869c886-bnmwq  1/1     Running   1 (13d ago)   17d
kubernetes-dashboard   kubernetes-dashboard-web-6ccf8d967-4rhgn              1/1     Running   1 (13d ago)   17d
```

Kubernetes Dashboard Installation

Add the Kubernetes Dashboard Helm Repository:

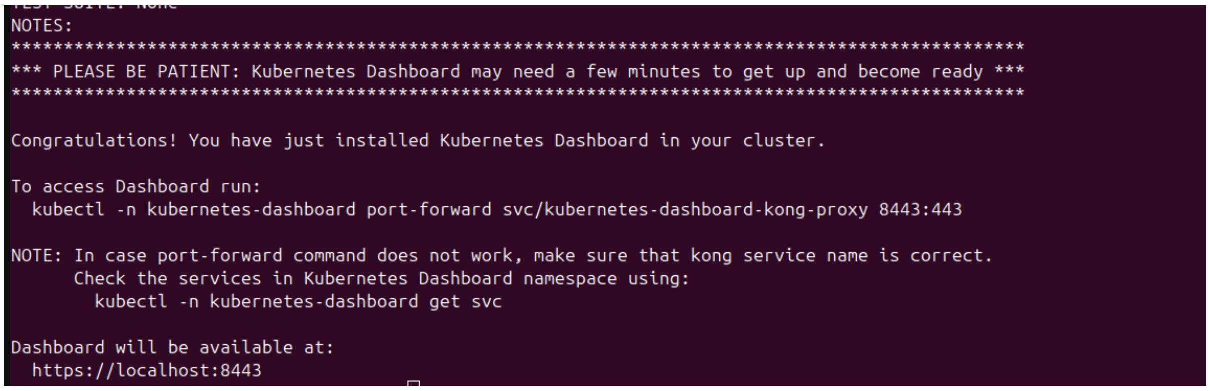
```
helm repo add kubernetes-dashboard
https://kubernetes.github.io/dashboard
```



```
"kubernetes-dashboard" already exists with the same configuration, skipping
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$ helm upgrade --install kubernetes-dashboard kubernetes-dashboard/kubernetes-dashboard \
  --create-namespace --namespace kubernetes-dashboard
Release "kubernetes-dashboard" does not exist. Installing it now.
NAME: kubernetes-dashboard
LAST DEPLOYED: Wed Dec 11 14:44:02 2024
NAMESPACE: kubernetes-dashboard
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
*************************************************************************************************
*** PLEASE BE PATIENT: Kubernetes Dashboard may need a few minutes to get up and become ready ***
*************************************************************************************************

Congratulations! You have just installed Kubernetes Dashboard in your cluster.

To access Dashboard run:
  kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy 8443:443

NOTE: In case port-forward command does not work, make sure that kong service name is correct.
      Check the services in Kubernetes Dashboard namespace using:
```

Install the Kubernetes Dashboard:

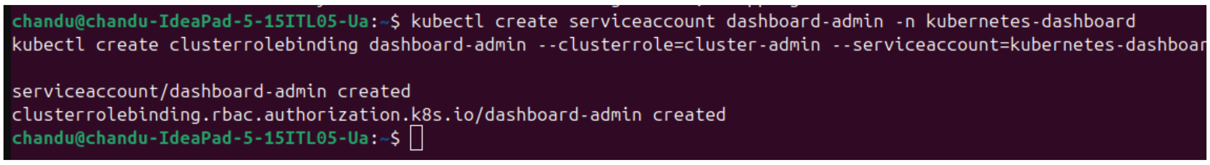
```
helm upgrade --install kubernetes-dashboard kubernetes-dashboard/kubernetes-dashboard \

  --create-namespace --namespace kubernetes-dashboard
```



```
NOTES:
*********************************************************************************************
*** PLEASE BE PATIENT: Kubernetes Dashboard may need a few minutes to get up and become ready ***
*********************************************************************************************

Congratulations! You have just installed Kubernetes Dashboard in your cluster.

To access Dashboard run:
  kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-proxy 8443:443

NOTE: In case port-forward command does not work, make sure that kong service name is correct.
      Check the services in Kubernetes Dashboard namespace using:
        kubectl -n kubernetes-dashboard get svc

Dashboard will be available at:
  https://localhost:8443
```

Create an Admin Account:

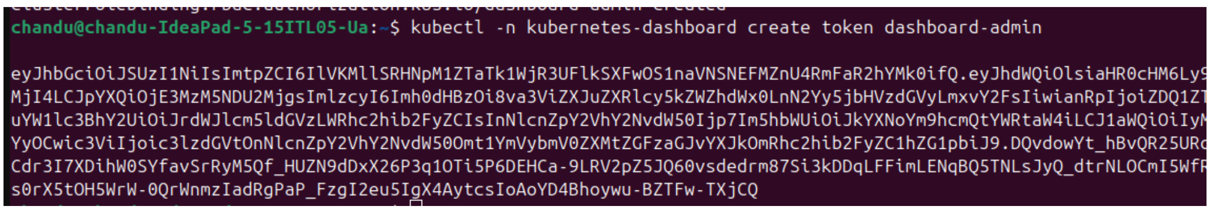
```
kubectl create serviceaccount dashboard-admin -n kubernetes-dashboard

kubectl create clusterrolebinding dashboard-admin \

  --clusterrole=cluster-admin \

  --serviceaccount=kubernetes-dashboard:dashboard-admin
```



```
chandu@chandu-IdeaPad-5-15ITL05-Ua:~$ kubectl create serviceaccount dashboard-admin -n kubernetes-dashboard
kubectl create clusterrolebinding dashboard-admin --clusterrole=cluster-admin --serviceaccount=kubernetes-dashboar

serviceaccount/dashboard-admin created
clusterrolebinding.rbac.authorization.k8s.io/dashboard-admin created
chandu@chandu-IdeaPad-5-15ITL05-Ua:~$
```

Generate a Token for Authentication:

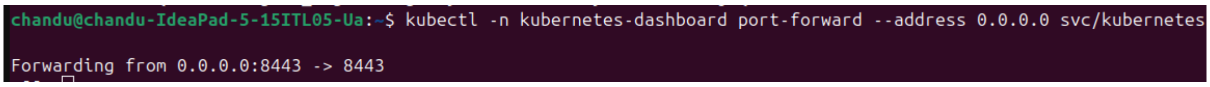
```
kubectl -n kubernetes-dashboard create token dashboard-admin
```



```
chandu@chandu-IdeaPad-5-15ITL05-Ua:~$ kubectl -n kubernetes-dashboard create token dashboard-admin

eyJhbGciOiJSUzI1NiIsImtpZCI6IlVKMllSRHNpM1ZTaTk1WjR3UFlkSXFwOS1naVNSNEFMZnU4RmFaR2hYMk0ifQ.eyJhdWQiOlsiaHR0cHM6Ly
MjI4LCJpYXQiOjE3MzM5NDU2MjgsImlzcyI6Imh0dHBzOi8va3ViZXJuZXRlcy5kZWZhdWx0LnN2Yy5jbHVzdGVyLmxvY2FsIiwianRpIjoiZDQ1Z
uYW1lc3BhY2UiOiJrdWJlcm5ldGVzLWRhc2hib2FyZCIsInNlcnZpY2VhY2NvdW50Ijp7Im5hbWUiOiJkYXNoYm9hcmQtYWRtaW4iLCJ1aWQiOiIy
YyOCwic3ViIjoic3lzdGVtOnNlcnZpY2VhY2NvdW50Omt1YmVybmV0ZXMtZGFzaGJvYXJkOmRhc2hib2FyZC1hZG1pbiJ9.DQvdowYt_hBvQR25UR
Cdr3I7XDihW0SYfavSrRyM5Qf_HUZN9dDxX26P3q1OTi5P6DEHCa-9LRV2pZ5JQ60vsdedrm87Si3kDDqLFFimLENqBQ5TNLsJyQ_dtrNLOCmI5Wf
s0rX5tOH5WrW-0QrWnmzIadRgPaP_FzgI2eu5IgX4AytcsIoAoYD4Bhoywu-BZTFw-TXjCQ
```

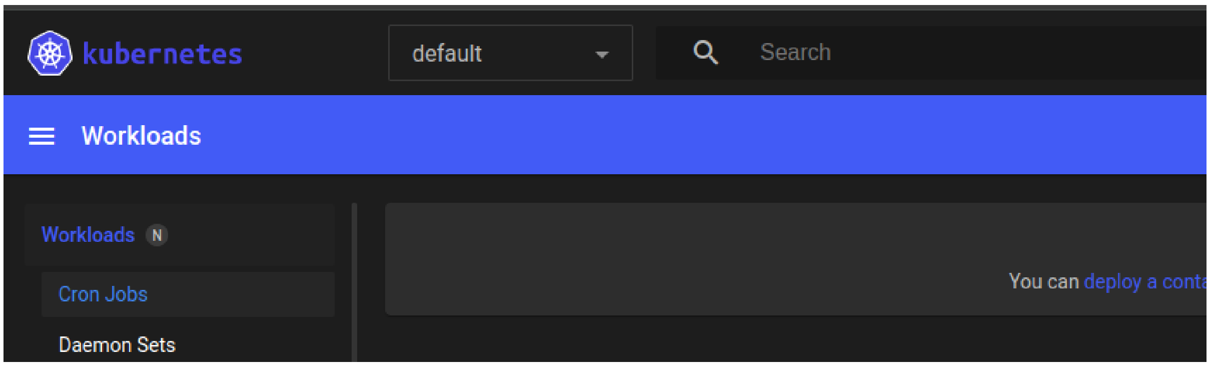
Access the Dashboard: Start the proxy:

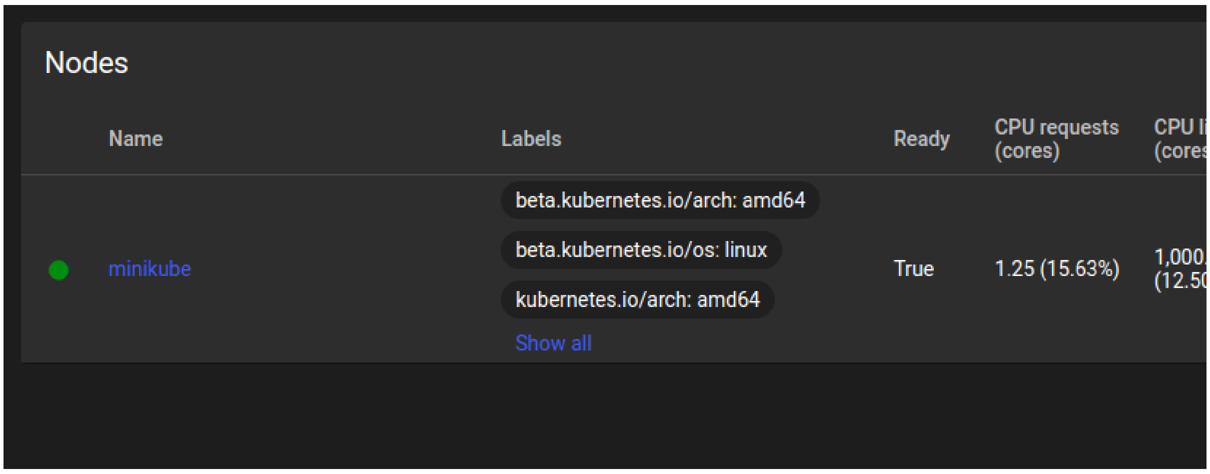
```
kubectl -n kubernetes-dashboard port-forward --address 0.0.0.0 svc/kubernetes-dashboard-kong-proxy 8443:443
```



```
chandu@chandu-IdeaPad-5-15ITL05-Ua:~$ kubectl -n kubernetes-dashboard port-forward --address 0.0.0.0 svc/kubernetes

Forwarding from 0.0.0.0:8443 -> 8443
```

Open the dashboard in a browser at https://<your-ip>:8443.





Nodes

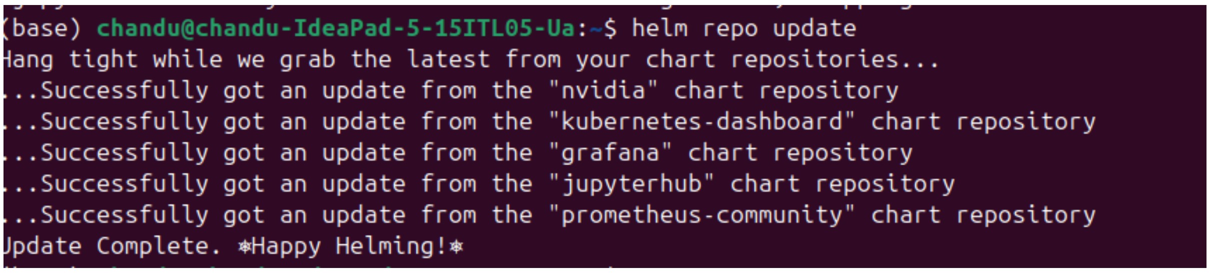
	Name	Labels	Ready	CPU requests (cores)	CPU li (cores
●	minikube	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/os: linux kubernetes.io/arch: amd64 Show all	True	1.25 (15.63%)	1,000 (12.50

JupyterHub Integration

Install JupyterHub Using Helm

Add the JupyterHub Helm Repository:

```
helm repo add jupyterhub https://jupyterhub.github.io/helm-chart/

helm repo update
```



```
(base) chandu@chandu-IdeaPad-5-15ITL05-Ua:~$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "nvidia" chart repository
...Successfully got an update from the "kubernetes-dashboard" chart repository
...Successfully got an update from the "grafana" chart repository
...Successfully got an update from the "jupyterhub" chart repository
...Successfully got an update from the "prometheus-community" chart repository
Update Complete. ⎈Happy Helming!⎈
```

Create a Configuration File (jupyterhub-config.yaml):

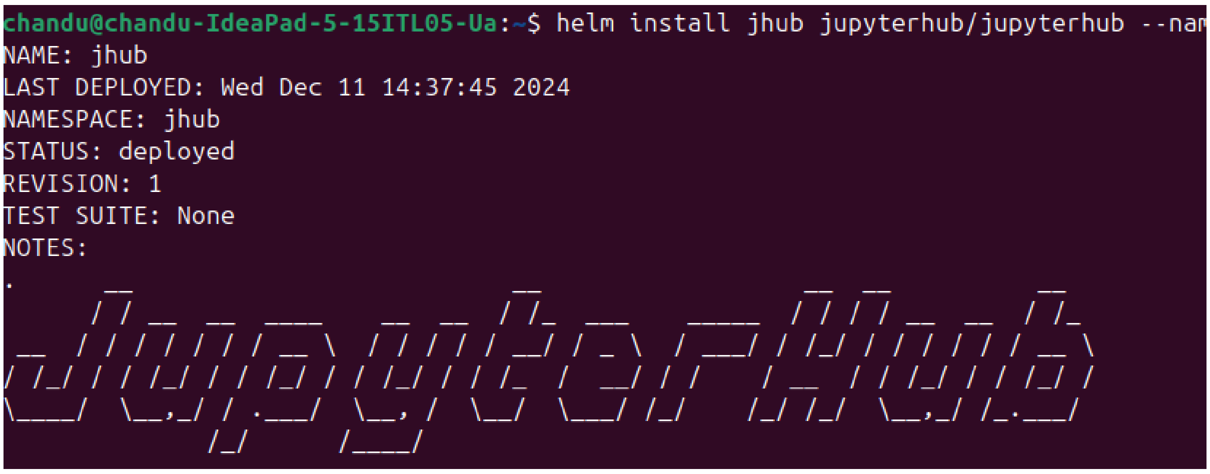
```
hub:
 config:
  Spawner:
   cpu_limit: 1
   mem_limit: '4G'
  JupyterHub:
   auth:
    type: dummy
    dummy:
     password: 'set-a-secure-password'

 extraConfig:
  proxy:
   secretToken: "generate-a-secret-token-here"

singleuser:
 extraResource:
  guarantees:
   cpu: "1"
   memory: 4Gi
```

```
    limits:
      cpu: "4"
      memory: 16Gi
      nvidia.com/gpu: "1"
  profileList:
    - display_name: "Small Instance"
      description: "Minimum resources for light workloads"
      kubespawner_override:
        cpu_guarantee: 1
        cpu_limit: 1
        mem_guarantee: "4G"
    - display_name: "GPU Instance"
      description: "Instance with GPU"
      kubespawner_override:
        cpu_guarantee: 1
        cpu_limit: 1
        mem_guarantee: "4G"
        mem_limit: "4G"
        extra_resource_limits:
          nvidia.com/gpu: "0"

proxy:
  secretToken: "generate-a-secret-token-here"


scheduling:
  userScheduler:
    enabled: true
  podPriority:
    enabled: true
  userPlaceholder:
    enabled: true
    replicas: 2
```

```
cull:
  enabled: true
  timeout: 3600
  every: 300

singleuser:
  storage:
    type: dynamic
    dynamic:
      storageClass: local-storage
    capacity: 4Gi
```

Deploy JupyterHub using helm

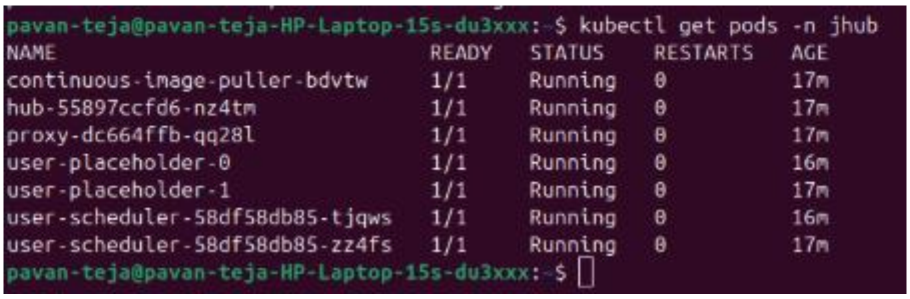
```
helm install jhub jupyterhub/jupyterhub \
  --namespace jhub \
  --create-namespace \
  --version=3.3.8 \
  --values jupyterhub-config.yaml
```



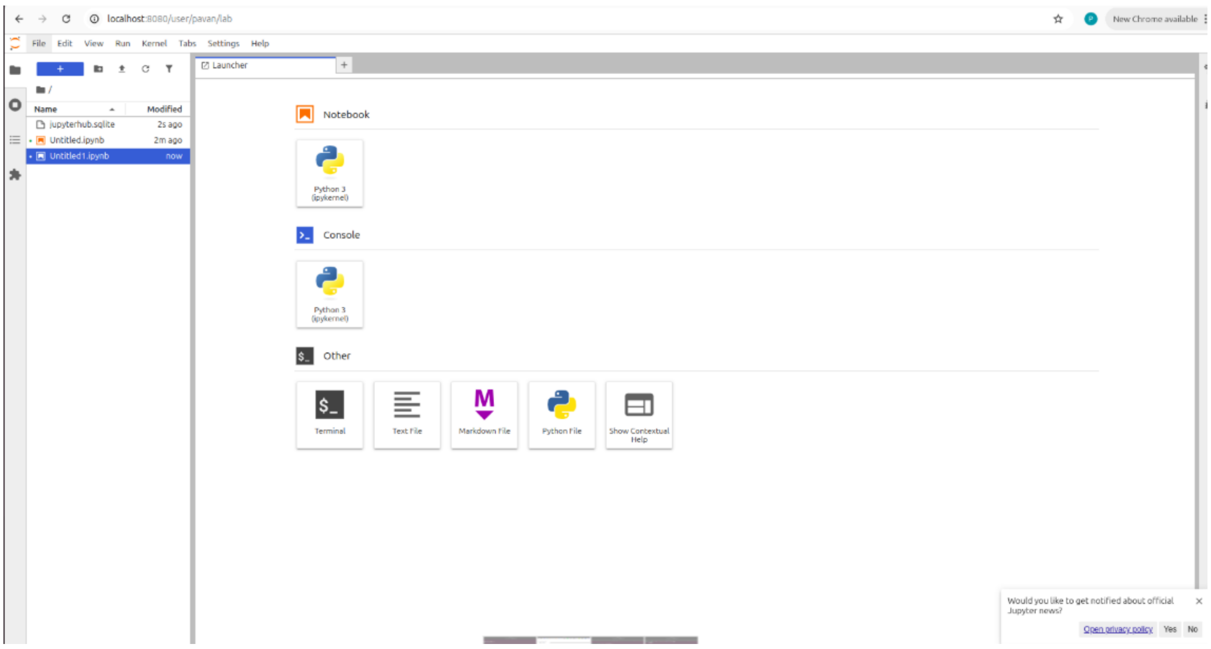
```
chandu@chandu-IdeaPad-5-15ITL05-Ua:~$ helm install jhub jupyterhub/jupyterhub --nam
NAME: jhub
LAST DEPLOYED: Wed Dec 11 14:37:45 2024
NAMESPACE: jhub
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
.      __                          __                  __  __          __
      / / __  __  ____    __  __  / /_  ___    _____  / / / / __  __  / /_
 __  / / / / / / / __ \  / / / / / __/ / _ \  / ___/ / /_/ / / / / / / __ \
/ /_/ / / /_/ / / /_/ / / /_/ / / /_  /  __/ / /    / __  / / /_/ / / /_/ /
\____/  \__,_/ / .___/  \__, /  \__/  \___/ /_/    /_/ /_/  \__,_/ /_.___/
              /_/      /____/
```

Verify Deployment:

```
kubectl get pods -n jhub
```

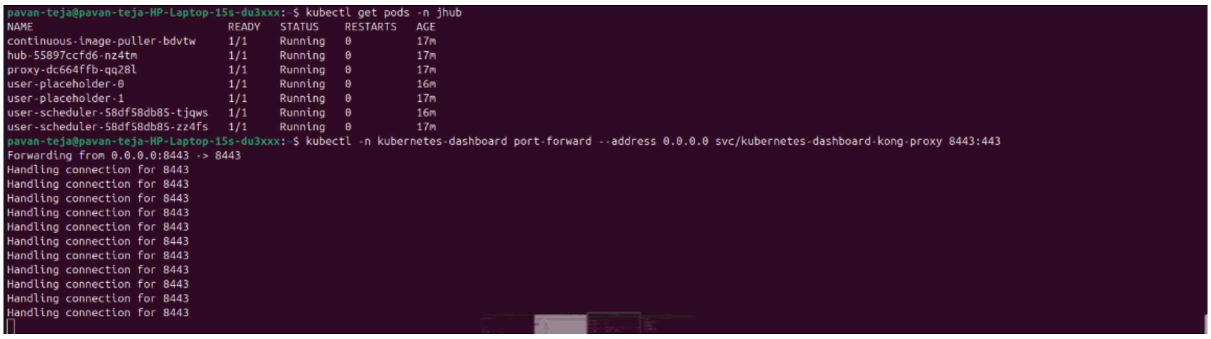


```
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$ kubectl get pods -n jhub
NAME                               READY   STATUS    RESTARTS   AGE
continuous-image-puller-bdvtw      1/1     Running   0          17m
hub-55897ccfd6-nz4tm               1/1     Running   0          17m
proxy-dc664ffb-qq28l               1/1     Running   0          17m
user-placeholder-0                 1/1     Running   0          16m
user-placeholder-1                 1/1     Running   0          17m
user-scheduler-58df58db85-tjqws    1/1     Running   0          16m
user-scheduler-58df58db85-zz4fs    1/1     Running   0          17m
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$
```

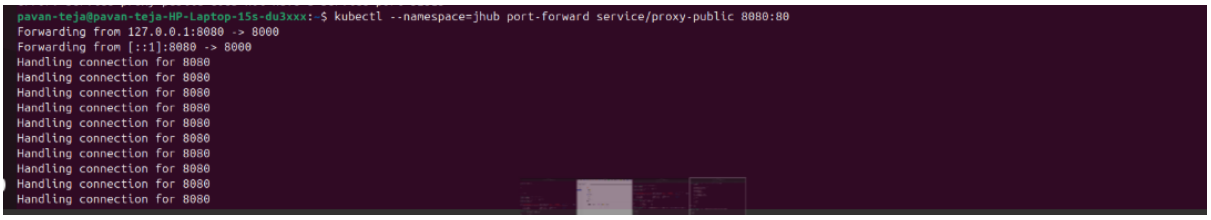


Name	Modified
jupyterhub.sqlite	2s ago
Untitled.ipynb	2m ago
Untitled1.ipynb	now

Would you like to get notified about official Jupyter news?



```
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$ kubectl get pods -n jhub
NAME                              READY   STATUS    RESTARTS   AGE
continuous-image-puller-bdvtw     1/1     Running   0          17m
hub-55897ccfd6-nz4tm              1/1     Running   0          17m
proxy-dc664ffb-qq28l              1/1     Running   0          17m
user-placeholder-0                1/1     Running   0          16m
user-placeholder-1                1/1     Running   0          17m
user-scheduler-58df58db85-tjqws   1/1     Running   0          16m
user-scheduler-58df58db85-zz4fs   1/1     Running   0          17m
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$ kubectl -n kubernetes-dashboard port-forward --address 0.0.0.0 svc/kubernetes-dashboard-kong-proxy 8443:443
Forwarding from 0.0.0.0:8443 -> 8443
Handling connection for 8443
Handling connection for 8443
Handling connection for 8443
Handling connection for 8443
Handling connection for 8443
Handling connection for 8443
Handling connection for 8443
Handling connection for 8443
Handling connection for 8443
Handling connection for 8443
Handling connection for 8443
```



```
pavan-teja@pavan-teja-HP-Laptop-15s-du3xxx:~$ kubectl --namespace=jhub port-forward service/proxy-public 8080:80
Forwarding from 127.0.0.1:8080 -> 8000
Forwarding from [::1]:8080 -> 8000
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
Handling connection for 8080
```

Troubleshooting

Pods Stuck in Pending State:

```
kubectl describe pod <pod-name> -n <namespace>
```

Networking Issues:

```
kubectl get pods -n kube-system | grep flannel
```

Logs for Errors:

```
kubectl logs <pod-name> -n <namespace>
```

Scaling JupyterHub: For better performance and scalability, you can adjust resource limits, spawn more user pods, or use a more powerful Kubernetes node setup. These adjustments can be made in the jupyterhub-config.yaml file and redeployed.

GPU Monitoring: In Kubernetes involves installing the NVIDIA GPU device plugin to track GPU usage. Prometheus collects GPU metrics like memory and usage, and Grafana displays these metrics on a dashboard. This helps in managing and monitoring GPU resources effectively in the cluster.

Multi-Tenant Access Control: Multi-tenant access control in Kubernetes ensures that different users or teams (tenants) can securely access and use shared resources. This is achieved by setting up role-based access control (RBAC) to define permissions for each tenant, allowing them to access only their designated resources. Using namespaces, tenants are isolated from each other, ensuring data security and preventing unauthorized access to resources across tenants.

Challenges and Solutions

Resource Constraints: Resource constraints in Kubernetes help manage and allocate CPU, memory, and other resources efficiently. By setting limits and requests for CPU and memory in pod configurations, Kubernetes ensures that each pod gets the necessary resources while preventing one pod from consuming excessive resources, which could affect others. This ensures optimal performance and resource utilization across the cluster.

Software Compatibility Issues: Software compatibility issues in Kubernetes arise when different software components, such as applications, libraries, or container images, do not work well together due to version mismatches or unsupported configurations. These issues can cause failures in deployment, resource allocation, or functionality. To avoid compatibility issues, it is essential to ensure that all components are compatible with the Kubernetes version, operating system, and other dependencies in use, and to regularly update software to maintain compatibility.

Network and Security Considerations: Network and security considerations in Kubernetes are crucial for ensuring that the cluster remains secure and performs efficiently. For network security, it’s essential to configure proper network policies to control traffic between pods and services. Kubernetes supports encryption, secure communication channels, and role-based access control (RBAC) to prevent unauthorized access. Additionally, securing sensitive data with tools like Secrets Management and using firewalls to limit external access to the cluster are essential practices for maintaining security while optimizing network performance.

Testing and Results

- **Test Cases**: Test cases for Kubernetes setup and configurations ensure that the system behaves as expected under various conditions. They can cover scenarios like pod creation, network connectivity, resource allocation, and user access control. Example test cases include:

- **Pod Deployment**: Verify that a pod is deployed successfully with specified resource requests and limits.
- **Service Accessibility**: Test if services are accessible from within and outside the cluster.
- **GPU Resource Allocation**: Ensure GPU resources are correctly allocated to pods and that Prometheus collects GPU metrics.
- **Role-Based Access Control (RBAC)**: Confirm that users with different roles have appropriate access to resources in their namespaces.
- **Network Policies**: Check if network policies prevent or allow traffic between pods as defined.
- **Performance Metrics**: Performance metrics in Kubernetes help monitor the efficiency and health of the cluster and its workloads. Key performance metrics include:
- **CPU and Memory Utilization**: Tracks the CPU and memory usage of pods, nodes, and containers to ensure resources are appropriately allocated and utilized.
- **Pod Health**: Measures pod status, including uptime, failures, and restarts, to ensure workloads are stable.
- **Network Throughput**: Monitors the amount of data transmitted and received across the cluster to ensure network performance.
- **GPU Utilization**: For clusters with GPU resources, monitoring GPU utilization helps ensure that workloads using GPUs are running efficiently.
- **Latency and Response Time**: Measures the time it takes for services or applications to respond to requests, indicating the responsiveness of the cluster.
- **Usability Results:** The Kubernetes cluster with JupyterHub integration provided a smooth deployment and efficient multi-tenant environment. Users found the interface intuitive, with easy access controls and effective resource monitoring, ensuring optimal performance and usability.

Detailed Chronology and Work Done

Initial Project Scope

What Was Done:

- Defined project goals and deliverables.
- Discussed the necessity of GPU clustering and multi-tenant utilization.

- Agreed upon a system resembling Jupyter Notebooks or Google Colab for user access.

Challenges/Errors:

None, as this was an exploratory discussion.

GPU Resources and Setup

What Was Done:

- Confirmed system specifications (Sneha-Inspiron-3593 with 2GB GPU and Chandu-IdeaPad-5).
- Decided to use an economical NVIDIA MX350 GPU for small-scale testing.
- Installed CUDA Toolkit version 12.6 to enable GPU computations.

Challenges/Errors:

- Limited GPU memory (2GB) posed constraints for testing large-scale clustering.
- Encountered difficulties in CUDA installation due to unsupported dependencies on one laptop.

What Was Done:

- Installed kubectl (Client v1.31.2, Server v1.31.0) for Kubernetes management.
- Set up Docker for containerization and created Kubernetes clusters.
- Discussed using a Kubernetes container with JupyterHub for GPU monitoring and multi-user access.

Challenges/Errors:

- Initial Kubernetes setup failed due to a mismatch in kubectl client and server versions.
- Resolved by updating both to the compatible versions.

GPU Clustering System

What Was Done:

- Implemented GPU resource discovery using NVIDIA utilities (nvidia-smi) within Kubernetes pods.
- Tested cluster creation and resource allocation among multiple tenants.
- Configured GPU monitoring through nvidia-docker for seamless GPU usage inside containers.

Challenges/Errors:

- nvidia-smi failed to detect GPUs inside the container due to missing drivers.
- Installed the required NVIDIA drivers manually to resolve the issue.

Web UI Integration

What Was Done:

- Integrated JupyterHub as the primary interface for GPU resource access.
- Designed a login-based system for user authentication and session management.
- Configured JupyterHub to allow users to select GPU resources for their notebooks.

Challenges/Errors:

- JupyterHub's integration with Kubernetes required additional configuration for persistent storage.
- Encountered permission issues when mounting storage volumes; resolved by updating Kubernetes role bindings.

Multi-Tenant Access and Scalability

What Was Done:

- Set up Kubernetes namespaces for isolating user workloads.
- Configured resource quotas to limit GPU usage per tenant.
- Implemented load balancing to optimize resource allocation.

Challenges/Errors:

- Resource quotas initially misconfigured, leading to some users exceeding GPU limits.
- Fixed by adjusting Kubernetes resource definitions and testing extensively.

Testing and Validation (November 2024)

What Was Done:

- Tested the tool's ability to handle multiple users accessing GPUs concurrently.
- Verified the monitoring dashboard to display real-time GPU usage.
- Conducted stress tests on the system to ensure stability under heavy workloads.

Challenges/Errors:

- Performance degraded under high concurrent user loads.
- Identified bottlenecks in the load balancer and optimized configurations.

Error Log and Solutions

Error/Challenge	Description	Solution
CUDA Installation Issues	Missing dependencies on some systems.	Updated dependencies and reinstalled CUDA Toolkit.
Kubernetes Client-Server Version Mismatch	Kubernetes failed to start due to incompatible versions.	Updated both client and server versions to v1.31.x.
Missing NVIDIA Drivers in Containers	nvidia-smi failed to detect GPUs inside Docker containers.	Installed NVIDIA drivers and configured nvidia-docker.
JupyterHub Volume Mount Errors	Persistent storage for Jupyter notebooks caused permission issues.	Adjusted Kubernetes role bindings and storage class configurations.
Resource Quota	GPU usage quotas exceeded by some	Corrected Kubernetes namespace and resource definitions.

Misconfiguration	users, leading to over-allocation.	
Load Balancing Bottlenecks	Performance degradation under high concurrent user workloads.	Optimized Kubernetes load balancer settings and tested with simulated workloads.

Overcoming Challenges

Throughout the development process, several challenges were encountered and resolved:

- **Compatibility Issues**: Addressed software and hardware compatibility challenges by ensuring proper configuration of NVIDIA drivers, Kubernetes, and JupyterHub.
- **Resource Constraints**: Optimized configurations for small-scale environments with limited GPU memory while maintaining scalability for larger setups.
- **Load Balancing and Quotas**: Fine-tuned Kubernetes resource definitions to prevent resource over-allocation and ensure equitable distribution.

Impacts and Implications

The successful implementation of this tool has broad implications for GPU resource management in multi-user environments:

- **Research and Development**: Facilitates collaborative research by enabling multiple users to access GPU resources securely and efficiently.
- **Educational Institutions**: Provides a platform for students and educators to run GPU-intensive simulations and experiments without requiring individual hardware setups.
- **Corporate Applications**: Empowers organizations to optimize GPU usage, reducing operational costs and enhancing productivity.

Conclusion

The **GPU Clustering and Multi-Tenant Utilization Tool** project represents a significant advancement in the domain of resource management for GPU-intensive computing tasks. By addressing the challenges of GPU under-utilization, user isolation, and scalability in shared environments, this project delivers a robust, user-friendly, and efficient solution for managing shared GPU resources.

Project Outcomes

Enhanced Resource Utilization:

- The tool dynamically allocates GPU resources across multiple users and workloads, maximizing efficiency and preventing under-utilization.
- Intelligent scheduling mechanisms ensure fair distribution of resources, even during peak demand.

User Isolation and Security:

- Secure role-based access control (RBAC) and Kubernetes namespaces provide user isolation, safeguarding workloads from interference.
- Multi-tenancy ensures equitable access to GPUs while maintaining strict resource boundaries between users.

Scalable and Intuitive System Design:

- Kubernetes serves as the backbone of the system, offering scalability, fault tolerance, and seamless load balancing for concurrent workloads.
- JupyterHub integration enables an intuitive interface, familiar to users of platforms like Google Colab, ensuring ease of adoption by researchers, developers, and educators.

Real-Time Monitoring and Management:

- GPU monitoring via NVIDIA plugins and integrated dashboards provides visibility into usage patterns, health metrics, and performance.
- The system empowers administrators to proactively address issues, optimize resource allocation, and maintain cluster health.

Future-Ready Infrastructure:

- Designed to be extensible, the tool can integrate advanced features such as AI-driven resource predictions, enhanced security policies, and cloud-based deployment for broader accessibility.

Final Thoughts

This project exemplifies how modern technologies like Kubernetes, NVIDIA GPU Operator, and JupyterHub can be integrated to solve real-world challenges in GPU resource management. The tool's modular architecture, combined with its scalability and user-friendly design, positions it as a valuable solution for research labs, universities, and industries requiring high-performance computing resources. By addressing key gaps in resource sharing, isolation, and monitoring, the GPU Clustering and Multi-Tenant Utilization Tool significantly enhances efficiency, accessibility, and collaboration in GPU-intensive environments.

Summary of Achievements

- **Comprehensive GPU Management**: Successfully deployed a Kubernetes-based system that clusters GPUs, enabling dynamic resource allocation and efficient utilization.
- **User-Friendly Interface**: Implemented a web-based interactive interface via JupyterHub for seamless multi-user access.
- **Multi-Tenant Support**: Ensured secure, isolated, and equitable GPU resource sharing through role-based access control and resource quotas.
- **Scalability and Load Balancing**: Tested and optimized the system for handling multiple users and workloads simultaneously, ensuring robust performance.

Future Scope

Potential Enhancements: Possible improvements include adding better security with advanced login methods, automatically adjusting resources based on demand, improving GPU use for heavy tasks, using better tools for monitoring performance, and making the network more secure with stronger policies.

Cloud-Based Deployment: cloud-based deployment allows for scaling resources easily and accessing services from anywhere. By moving the Kubernetes cluster and JupyterHub to the cloud, you can ensure better flexibility, high availability, and easier management. Popular cloud platforms like AWS, Google Cloud, or Azure can offer the necessary infrastructure, and they also provide tools for managing resources, networking, and security more effectively. Cloud services can further enhance performance, making it easier to handle large workloads and ensure seamless access for users across different locations.

AI Integration: AI integration can improve the system by automating tasks, making better decisions, and improving resource use. It can help predict when more resources are needed, detect problems early, and analyze data faster. AI can also be used to personalize services and monitor system performance in real-time, making everything run more efficiently.

References

1. Kubernetes Documentation. "Kubernetes Official Documentation." https://kubernetes.io/docs/
2. NVIDIA Corporation. "NVIDIA CUDA Toolkit Documentation." https://docs.nvidia.com/cuda/
3. Docker Inc. "Docker Documentation." https://docs.docker.com/
4. Helm. "Helm Documentation." https://helm.sh/docs/
5. JupyterHub. "JupyterHub Official Documentation." https://jupyterhub.readthedocs.io/
6. NVIDIA Corporation. "NVIDIA GPU Operator." https://github.com/NVIDIA/gpu-operator
7. Prometheus. "Prometheus Monitoring System and Time Series Database." https://prometheus.io/docs/
8. Grafana Labs. "Grafana Documentation." https://grafana.com/docs/
9. CoreOS. "Flannel Networking for Kubernetes." https://github.com/flannel-io/flannel
10. Google. "Kubectl Reference Documentation." https://kubernetes.io/docs/reference/kubectl/
11. Red Hat. "Role-Based Access Control (RBAC) in Kubernetes." https://kubernetes.io/docs/reference/access-authn-authz/rbac/

12. NVIDIA Corporation. "NVIDIA GPU Cloud (NGC)." https://www.nvidia.com/en-us/gpu-cloud/
13. Apache Foundation. "Apache Airflow for Workflow Automation." https://airflow.apache.org/
14. Medium Article. "Kubernetes GPU Scheduling with NVIDIA Device Plugins." https://medium.com/
15. GitHub. "Kubernetes Helm Charts for JupyterHub." https://github.com/jupyterhub/helm-chart
16. OpenAI. "High-Performance Computing Using Kubernetes." https://openai.com/
17. Argo Project. "Argo Workflows for Kubernetes." https://argoproj.github.io/
18. Intel. "Best Practices for Using Kubernetes with GPU Workloads." https://www.intel.com/
19. Microsoft. "Azure Kubernetes Service with NVIDIA GPUs." https://learn.microsoft.com/
20. CNF Testbed. "Scalable Kubernetes Cluster Setup for GPUs." https://github.com/cncf/cnf-testbed
21. Kubernetes Blog. "Monitoring GPU Performance in Kubernetes Clusters." https://kubernetes.io/blog/
22. NVIDIA Developer Blog. "Accelerating Kubernetes with NVIDIA GPUs." https://developer.nvidia.com/blog/
23. ResearchGate. "GPU Clustering for Multi-Tenant Cloud Environments." https://www.researchgate.net/
24. SpringerLink. "Dynamic Resource Allocation in Kubernetes." https://link.springer.com/
25. Wiley Online Library. "GPU Monitoring Techniques for Kubernetes Clusters." https://onlinelibrary.wiley.com/