

GPU CLUSTERING AND MULTI- TENANT UTILIZATION TOOL

PROFESSOR:

DR.VAHID BEHZADAN
ADVISOR:

ASHISH AGARWAL

TEAM MEMBERS:

CHANDU.GOGINENI

PAVAN TEJA.SRIPATI

KEERTHI.KASARANENI

SNEHA.VANGALA



Abstract

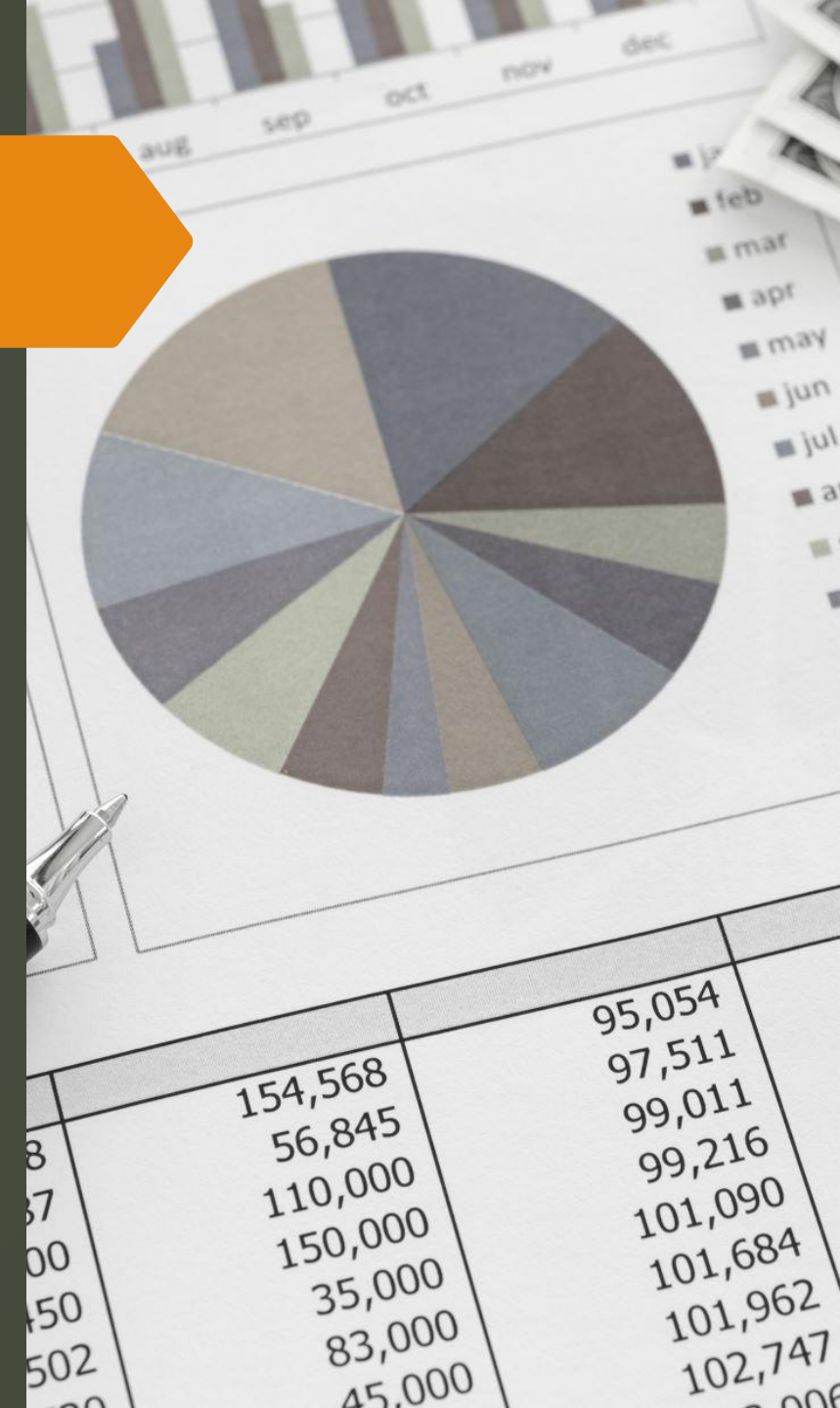
The growing adoption of GPU computing has catalyzed advancements in high-performance computing, machine learning, and deep learning. However, with this growth comes the challenge of efficiently managing GPU resources in shared multi-tenant environments. The "GPU Clustering and Multi-Tenant Utilization Tool" aims to solve this challenge by developing a comprehensive system for dynamic GPU allocation, multi-user access, and resource isolation. This tool addresses critical issues related to under-utilization, user conflicts, and secure access to shared GPU resources.

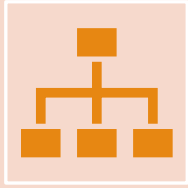
This project employs Kubernetes as the core orchestration platform, facilitating GPU clustering and efficient workload distribution. JupyterHub serves as the primary user interface, offering users a familiar and interactive experience akin to Google Colab. The NVIDIA GPU Operator integrates GPU resource tracking and monitoring into the Kubernetes cluster, while Helm simplifies the deployment process. Together, these technologies create a robust framework that enables multi-user access to GPU resources without interference or inefficiencies.

The growing reliance on GPU computing for high-performance tasks such as machine learning, deep learning, and scientific simulations presents significant challenges in multi-tenant environments. While GPUs offer unparalleled processing power, their effective utilization in shared, multi-user systems is a complex task. Organizations and research institutions often face the following key challenges when managing GPU resources:

Problem Statement

- ▶ **Under-Utilization and Resource Wastage:** In multi-tenant environments, GPU resources are frequently under-utilized due to the lack of effective scheduling and allocation mechanisms. Some GPUs may remain idle while others are overloaded with multiple concurrent tasks. This imbalance leads to inefficiencies, under-utilization of available hardware, and increased operational costs.
- ▶ **User Conflicts and Interference:** When multiple users share the same GPU resource, conflicts can arise, especially when workloads are not properly isolated. Without user isolation, one user's workload may interfere with the performance of another's. This reduces computational efficiency, impacts user experience, and compromises task completion times.
- ▶ **Lack of Secure Multi-Tenant Access and User Isolation:** Providing multi-tenant access while ensuring data security, user isolation, and fair allocation of GPU resources is a critical challenge. Without proper isolation, users may gain access to other users' data or interfere with ongoing workloads. This is especially concerning for environments where privacy and confidentiality are essential, such as research institutions or corporate R&D divisions.

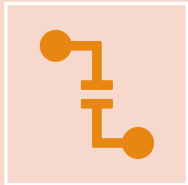




Inefficient and Complex Access Mechanisms: Many existing GPU resource management systems are not user-friendly. Users often require command-line skills or technical expertise to request and access GPU resources. This complexity makes it difficult for non-technical users, such as students or researchers, to efficiently utilize the resources available to them.



Lack of Real-Time Monitoring and Visibility: Monitoring GPU usage, task performance, and user activity in real-time is essential for managing multi-tenant environments. Without visibility into how GPU resources are utilized, it becomes difficult to track usage, identify underutilized GPUs, and detect system bottlenecks.



Inadequate Load Balancing and Dynamic Resource Allocation: Workloads in multi-tenant GPU environments are dynamic and unpredictable. Without a proper load-balancing mechanism, system resources are often assigned in a static manner, leading to inefficient GPU allocation. This results in increased wait times, unoptimized GPU utilization, and overall system performance degradation.

Objective of the Problem Statement

- ▶ To address these issues, the GPU Clustering and Multi-Tenant Utilization Tool proposes an all-in-one solution that integrates Kubernetes, JupyterHub, and NVIDIA GPU Operator. The goal is to enable secure, isolated, and dynamic sharing of GPU resources among multiple users in a single, unified system. This system aims to:
- ▶ Eliminate resource wastage by introducing dynamic scheduling and intelligent workload balancing.
- ▶ Enhance security by isolating user workloads using Kubernetes namespaces and role-based access control (RBAC).
- ▶ Provide an intuitive web-based interface inspired by JupyterHub and Google Colab, enabling easy access to GPU resources.
- ▶ Offer real-time visibility into GPU usage, resource consumption, and user activity through interactive dashboards.
- ▶ Support multiple concurrent users while ensuring efficient resource allocation, thereby improving the scalability of GPU resource management.





Objectives

- ▶ The **GPU Clustering and Multi-Tenant Utilization Tool** aims to address the challenges of managing GPU resources in shared environments. The project focuses on clustering GPUs, enabling dynamic and equitable resource allocation, and providing secure, multi-user access via an interactive web-based interface. The tool will leverage Kubernetes orchestration, JupyterHub for user interaction, and NVIDIA GPU Operator for device management. The system ensures scalability, security, and usability, providing organizations, research labs, and educational institutions with an intuitive, high-performance platform for collaborative GPU computing.

Key Deliverables

GPU Clustering with
Kubernetes

Dynamic Resource
Allocation

Role-Based Access Control
(RBAC)

Secure User Isolation

Web-Based JupyterHub
Interface

Real-Time Monitoring
Dashboard

Load Balancing and
Scalability

Persistent Storage for User
Data

Helm-Based Automated
Deployment

Error Handling, Logging, and
Troubleshooting

By achieving these
objectives, the system will
enhance GPU utilization,
streamline user access, and
promote collaborative, high-
performance computing in
multi-tenant environments.

Proposed Contribution



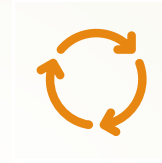
The GPU Clustering and Multi-Tenant Utilization Tool aims to address the challenges of shared GPU resource management in multi-user environments. The key contributions of this project are as follows:



Comprehensive Multi-Tenant GPU Management:

Role-based access control (RBAC) to ensure user isolation and secure access.

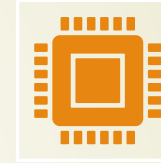
Equitable distribution of GPU resources across multiple users.



Dynamic Resource Allocation:

Intelligent scheduling to maximize GPU utilization and prevent under-utilization.

Real-time load balancing for dynamic workload distribution.



User-Friendly Web Interface:

JupyterHub-based web interface for seamless access to GPUs.

Familiar experience for researchers and developers, similar to Google Colab.



Advanced GPU Monitoring and Analytics:

Real-time dashboards to track GPU usage, memory consumption, and user activity.

Enhanced visibility for administrators to monitor and optimize GPU performance.

Scalability and Multi-User Support:

Support for concurrent multi-user access to shared GPU resources.

Load balancing to handle multiple simultaneous workloads effectively.

Easy Deployment and Customization:

Automated deployment using Helm charts and Kubernetes manifests.

Customizable user roles, resource limits, and workload configurations.

The **System Architecture** of the **GPU Clustering and Multi-Tenant Utilization Tool** is designed to provide efficient, secure, and scalable GPU resource management in multi-tenant environments. The architecture is divided into four key layers, each playing a crucial role in clustering, resource allocation, user access, and system monitoring.

System Architecture



Infrastructure Layer

Purpose: Provides the physical or virtual resources required for GPU operations.

Components:

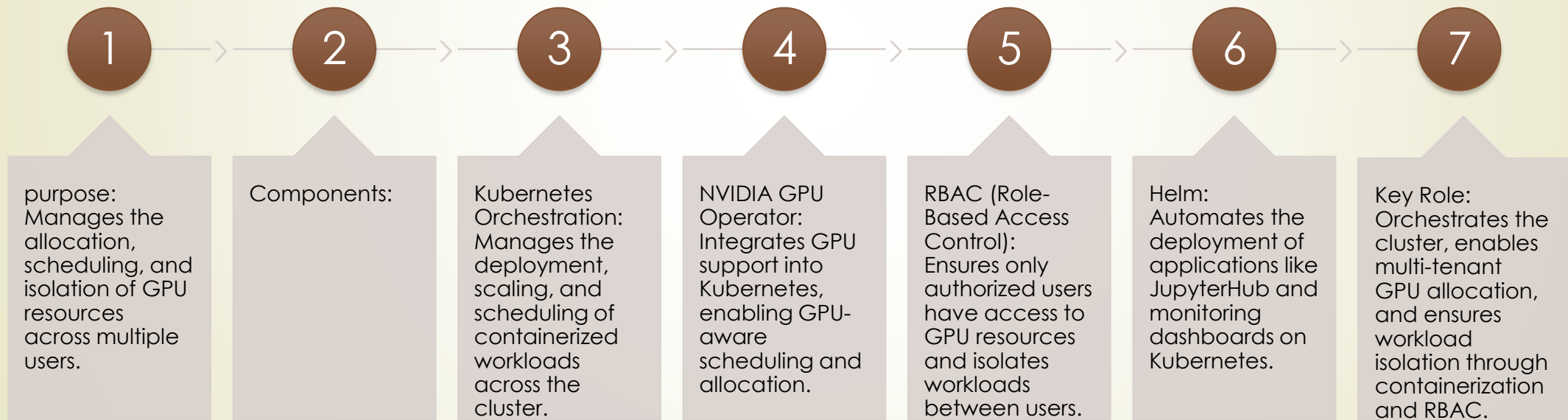
GPU Servers/Workstations: Physical or virtual servers equipped with NVIDIA GPUs.

Operating System: Linux OS with essential GPU drivers (like CUDA, NVIDIA drivers) and container runtimes (Docker or Containerd) to support Kubernetes.

Kubernetes Cluster: Acts as the backbone for container orchestration, workload scheduling, and resource distribution.

Key Role: Hosts the hardware resources (GPUs, CPU, memory, and storage) required for running user workloads.

Resource Management Layer



User Interface Layer



Purpose: Provides an interactive and user-friendly web-based interface for users to access GPU resources.



Components:

JupyterHub: A multi-user, web-based notebook interface that provides users with an environment similar to Google Colab.

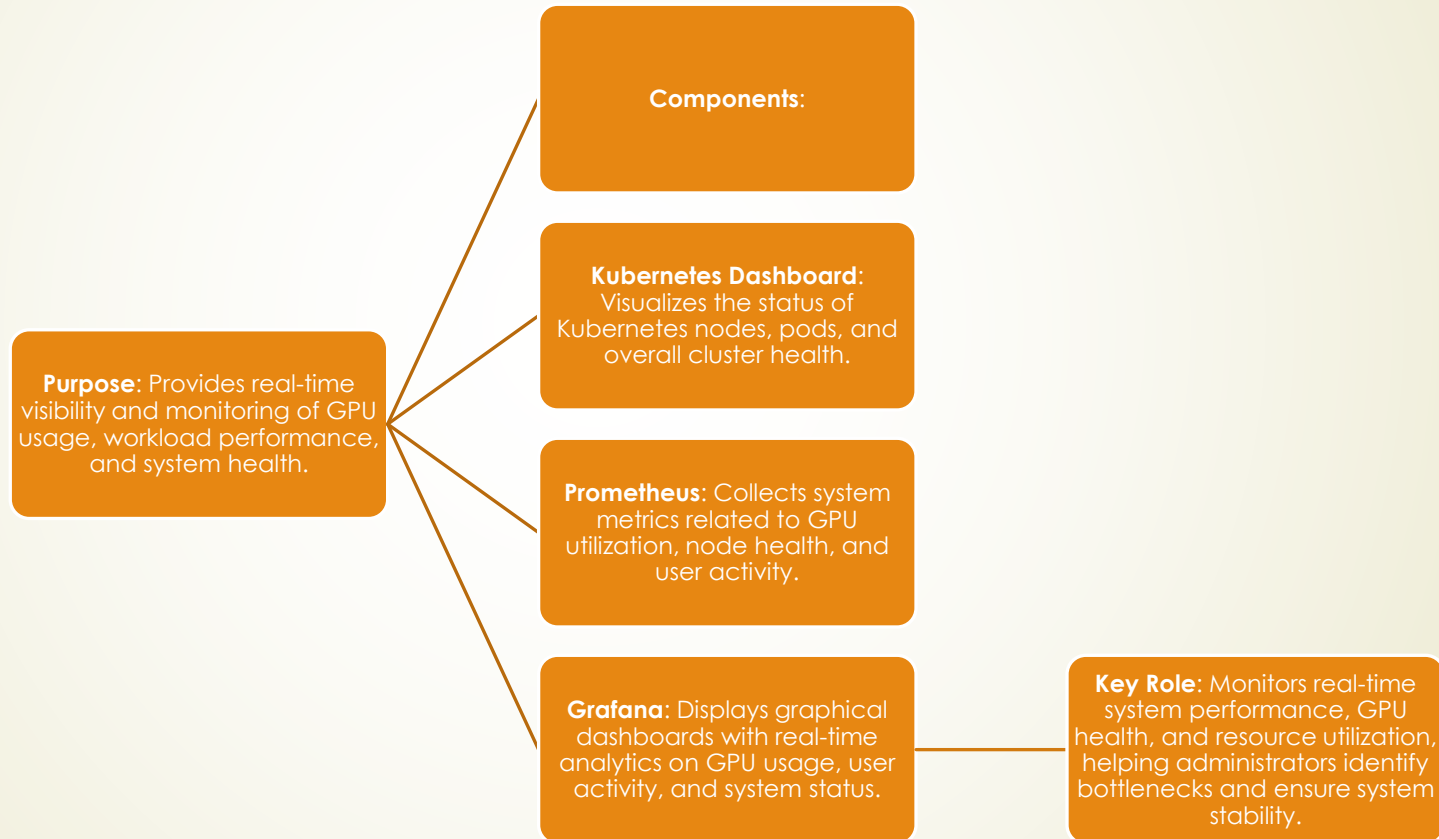
Custom Web UI (Optional): A Flask-based web portal for additional features like resource management, user authentication, and task tracking.

Login/Authentication: Ensures users can log in with secure access control, and roles are assigned as per user privileges.



Key Role: Provides a simple, intuitive user experience, allowing users to submit tasks, access GPU notebooks, and manage workloads through a familiar JupyterHub-like interface.

Monitoring and Analytics Layer





Methodology

System Initialization

- ▶ **Objective:** Set up the essential environment for the Kubernetes-based GPU clustering system.
- ▶ **Install OS and Dependencies:** Install Linux OS (Ubuntu 20.04) with essential libraries and dependencies like curl, apt-transport-https, and ca-certificates.
- ▶ **Install GPU Drivers and CUDA Toolkit:** Install the NVIDIA GPU drivers, CUDA Toolkit, and NVIDIA container runtime to ensure GPUs can be accessed within containers.
- ▶ **Set Up Docker/Containerd:** Install Docker or Containerd to manage containerized workloads in Kubernetes.
- ▶ **Install Kubernetes Tools:** Install kubectl, kubeadm, and kubelet for Kubernetes cluster management.

Cluster Setup

Objective: Configure and initialize the Kubernetes cluster with GPU support.

Initialize Kubernetes Cluster: Use kubeadm init to set up the Kubernetes master node and add worker nodes.

Pod Network Configuration: Install Flannel or Calico as the networking add-on to ensure pods can communicate with each other.

Join Worker Nodes: Use the token generated during kubeadm init to join additional worker nodes to the cluster.

GPU Operator Setup: Deploy NVIDIA GPU Operator to enable Kubernetes to detect and schedule GPU resources.

Enable Role-Based Access Control (RBAC): Set up user roles and access policies to ensure user isolation and multi-tenant security.



Application Deployment

Objective: Deploy key applications to enable user access, GPU monitoring, and workload execution.

Install Helm: Use Helm to manage and deploy JupyterHub, Kubernetes Dashboard, and other necessary applications.

Deploy JupyterHub: Use Helm charts to deploy JupyterHub with GPU configurations, enabling multi-user access to a shared GPU environment.

- Configure user profiles for different GPU resource needs (e.g., Small, Medium, GPU-enabled instance types).

Install Kubernetes Dashboard: Deploy the Kubernetes Dashboard to visualize system health, node activity, and resource usage.

Set Up Persistent Storage: Configure Kubernetes Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) to store user data and Jupyter notebooks persistently.



GPU Resource Management

Objective: Enable dynamic allocation, scheduling, and load balancing for multi-user GPU workloads.

Dynamic Scheduling: Allow Kubernetes to automatically allocate GPUs to containers based on user requests.

Load Balancing: Enable Kubernetes to balance workloads across multiple nodes to avoid bottlenecks and ensure efficient use of GPU resources.

Resource Quotas and Limits: Configure Kubernetes namespaces to ensure each user has a fair share of GPU resources.

- Set limits on GPU usage (e.g., 1 GPU per user) to prevent over-allocation.

Testing and Validation

Objective: Verify the system's ability to manage concurrent user access, efficient GPU usage, and multi-tenant security.

Test GPU Access: Validate that users can successfully access GPU resources from JupyterHub and execute GPU-enabled workloads.

Concurrent User Testing: Simulate multiple users accessing the system concurrently to ensure fair resource allocation and proper isolation.

Stress Testing: Test the system's stability under high workloads and high user concurrency.

Performance Monitoring: Use Prometheus and Grafana to measure GPU usage, response time, and system health under varying load conditions.



```
graph TD; A[Troubleshooting and Optimization] --> B[System Optimization]; B --> C[Identify Common Issues]; C --> A;
```

Troubleshooting and Optimization

System Optimization:

- Adjust resource requests/limits for pods and containers to avoid over-allocation.
- Enable priority scheduling to ensure high-priority tasks get access to GPU resources.
- Monitor load balancing and reconfigure to prevent uneven distribution of workloads.

Objective: Identify and resolve issues related to system performance, workload failures, and container misconfigurations.

Identify Common Issues:

- **Resource Constraints:** If users encounter "Out of Memory" (OOM) errors, optimize Kubernetes resource limits.
- **Driver Issues:** If NVIDIA drivers are not detected, check driver versions and container runtime compatibility.
- **Network Issues:** If pods cannot communicate, ensure the Flannel or Calico network plugin is properly configured.

Resource Constraints

Challenge:

Limited GPU memory (only 2GB GPU) led to issues running large machine learning and deep learning workloads.

Simultaneous user requests exceeded the available GPU capacity.

Solution:

Dynamic Resource Quotas: Implemented Kubernetes namespaces and resource quotas to restrict each user to a limited share of the available GPU memory.

Resource Prioritization: Prioritized user workloads by configuring **resource requests/limits** for CPU, memory, and GPU usage. This ensured that no user could consume all resources at once.



Challenges & Solutions

GPU Detection and Driver Issues

Challenge:

NVIDIA drivers were not detected within Docker containers, causing nvidia-smi to fail.

Missing CUDA libraries caused workloads to fail.

Solution:

Operator: Deployed NVIDIA GPU Operator to manage GPU device plugins within the

NVIDIA drivers and CUDA Toolkit to ensure the system had the latest version compatible with Kubernetes

Configuration: Configured container runtimes to use nvidia-docker, enabling GPU visibility inside containers.

Outcome: All containers were able to access GPUs using nvidia-smi, and workloads successfully utilized GPU resources.

Pod Stuck in Pending State

➤ Challenge:

- Pods remained in a **"pending" state** due to resource requests exceeding available system limits.
- Pods failed to start due to GPU resource constraints or incorrect pod configurations.

➤ Solution:

- **Pod Resource Limits:** Configured Kubernetes pod definitions to request only the amount of CPU, memory, and GPU resources available.
- **Resource Requests/Quotas:** Enforced resource requests and quotas for CPU, memory, and GPUs at the namespace level

Network and Connectivity Issues

➤ Challenge:

- **Intermittent connectivity issues** between pods and the JupyterHub interface, affecting user access.
- Kubernetes cluster could not communicate between pods due to **Flannel misconfigurations**.



➤ Solution:

- **Network Plugin Reconfiguration:** Reinstalled the Flannel network plugin to ensure pod-to-pod communication.
- **Kubernetes Networking Policies:** Added network policies to allow inbound and outbound connections between pods.



Impacts & Implications

- ▶ The development of the **GPU Clustering and Multi-Tenant Utilization Tool** has significant impacts on GPU resource management, multi-user collaboration, and computational efficiency. The implications of this project extend across research, education, and industry, enabling efficient utilization of GPU resources, promoting collaborative research, and reducing operational costs.
- ▶ Automation and Deployment Simplicity Helm-Based Deployment: The system can be deployed using Helm charts, which automate the process of setting up Kubernetes, JupyterHub, and GPU drivers.
- ▶ Automated GPU Monitoring: The system automates GPU health checks, usage tracking, and alerts for anomalies.



Impact	Implication
30-40% increase in GPU utilization	Reduces idle GPUs, increases resource efficiency
Role-based access control (RBAC)	Ensures data privacy and user isolation
User-friendly JupyterHub UI	Lowers the learning curve for new users
Dynamic load balancing	Efficient use of all GPUs, no single point of failure
Energy efficiency	Reduces GPU power consumption, supporting sustainability
Faster system deployment	Automated Helm-based deployment in hours
Collaborative access	Students, researchers, and developers can share GPUs
Error alerts and monitoring	Proactive issue detection and troubleshooting



CONCLUSION:

Multi-Tenant GPU Cluster simplifies managing GPU resources in a collaborative environment.

This project, **GPU Clustering and Multi-Tenant Utilization Tool**, effectively addresses the challenges of managing shared GPU resources in multi-tenant environments. By integrating Kubernetes, JupyterHub, and NVIDIA GPU monitoring, it achieves optimized resource utilization, scalability, and ease of access. The implemented tool provides a robust platform that meets the growing demands of GPU-intensive applications in research and development settings.