

CS777 Term Paper Submission – Venkata Sandeep Yerra

*Kubernetes: An Exploration of Its Evolution, Architecture, and Impact on Modern Cloud Computing and MLOps*

**Venkata Sandeep Yerra**  
**U32081493**  
**Boston University**

CS 777 Term Paper Submission

## Table of Contents

<b>Executive Summary.....</b>	<b>3</b>
<b>Introduction.....</b>	<b>4</b>
<b>Historical Background .....</b>	<b>4</b>
<b>Evolution of Containers and the Rise of Kubernetes .....</b>	<b>5</b>
<b>Kubernetes Architecture.....</b>	<b>6</b>
<b>Kubernetes in MLOps .....</b>	<b>8</b>
<b>Demo Objective .....</b>	<b>9</b>
<b>Demo Setup.....</b>	<b>9</b>
<b>Demo Architecture.....</b>	<b>9</b>
<b>Demo Environment Setup Code.....</b>	<b>10</b>
<b>Demo Deployment Screenshots.....</b>	<b>12</b>

## Executive Summary

Kubernetes, known as K8s, has redefined cloud computing by revolutionizing the way applications are deployed, managed, and scaled. It stemmed from Google's expertise in handling large container workloads with Borg and Omega, signifying a major industry shift towards containerization and microservices.

Borg and Omega, Google's early systems, were the precursors to Kubernetes. Borg was vital for task scheduling and resource allocation, while Omega addressed Borg's monolithic challenges by introducing flexible scheduling. Kubernetes was birthed in 2014 from the lessons of Borg and Omega, embracing an open design and combining it with popular container technologies like Docker.

Docker's introduction in 2013 popularized container technology. The adoption of containers spurred a move to microservices architecture, where applications are segmented into small, independent services. This rise of containerized microservices necessitated a tool like Kubernetes for efficient management at scale. Kubernetes evolved to manage containers effectively, drawing from Google's experiences with Borg and Omega.

Kubernetes architecture includes:

- Pods: Basic units housing one or several containers.
- Nodes: Worker machines, physical or virtual.
- Services: Defines a set of Pods and access policies.
- Deployment: Manages Pods and describes an application's life cycle.

It operates on a declarative model, focusing on the desired application state.

Kubernetes excels in orchestration, with capabilities like:

- Scheduling and Resource Allocation: Auto-scheduling based on resource availability.
- Health Checks and Self-healing: Auto-restarting or rescheduling of failed Pods.
- Load Balancing and Service Discovery: Distributes network traffic for stability.
- Scaling and Rollouts: Supports automated/manual scaling and application updates.
- Configuration and Secret Management: Safely manages configurations and sensitive information.
- Networking: Manages IP addresses, subnets, and port allocations.

Kubernetes is conducive to continuous integration and deployment practices, aligning with the DevOps approach for faster and more efficient release cycles. Kubernetes plays a vital role in Machine Learning Operations (MLOps), streamlining the management of ML workflows, from data processing to model deployment. Kubernetes ensures reproducible ML pipelines, which are critical for consistent model development.

## Introduction

Kubernetes, commonly known as K8s, has revolutionized cloud computing by transforming the deployment, management, and scaling of applications. Born from Google's expertise in managing massive container workloads with its systems Borg and Omega, Kubernetes reflects a significant shift in the tech industry towards containerization and microservices architecture. This shift influences how contemporary applications are structured and operated.

As an open-source container orchestration platform, Kubernetes simplifies and automates the tasks traditionally associated with deploying and managing containerized applications. Before Kubernetes, these processes were manual and error-prone. The advent of container technology, which packages applications with all their dependencies, changed the landscape, though it also added complexity, especially at scale. Kubernetes addresses this by automating deployment, scaling, and operations of application containers across clusters of hosts.

Kubernetes thrives on simplicity and automation, embracing a declarative approach for configuration and management. This approach means defining the desired state of applications, while Kubernetes handles the logistics to achieve and maintain that state. The platform's ecosystem extends beyond the technology itself, encompassing a broad range of tools and services for monitoring, networking, and security. Its widespread adoption across industries, from startups to large enterprises, underscores its role as a vital tool in modern cloud-native application management and the ongoing evolution of cloud infrastructure.

## Historical Background

### *The Genesis in Google's Borg and Omega*

Kubernetes traces its lineage to Google's pioneering work with Borg and Omega, systems that managed large-scale containerized applications long before "containers" became a buzzword. Borg, the forefather in this lineage, was instrumental in efficiently scheduling tasks, allocating resources, and ensuring high availability across Google's massive server clusters. This technology, serving as the backbone for major services like Google Search and Gmail, focused on scalability, reliability, and efficiency using containers for task isolation — a groundbreaking approach at the time.

However, Borg's monolithic structure led to challenges in flexibility and maintainability, setting the stage for Omega. Omega evolved from Borg, introducing more advanced and flexible scheduling mechanisms and shared-state scheduling models. This shift allowed multiple schedulers to operate concurrently, enhancing resource utilization and speeding up decisions.

### *From Borg and Omega to Kubernetes*

Drawing lessons from Borg and Omega, Kubernetes was conceived in 2014 with the goal of extending Google's internal achievements to a wider audience. It distinguished itself by adopting an open, modular design and using declarative configurations for component management. This modularity, combined with the integration of popular container technologies like Docker, made Kubernetes user-friendly, scalable, and highly adaptable to various environments.

Launched publicly in 2015, Kubernetes marked a significant milestone in container orchestration and cloud computing. It encapsulated Google's expertise in an open-source, community-fueled framework, making sophisticated cluster management accessible to a broader range of users and scenarios. Kubernetes not only inherited Borg and Omega's ethos of efficiency, scalability, and reliability but also amplified these qualities through its community-driven development and enhancements.

Kubernetes' journey from Google's internal systems to a global open-source project demonstrates its evolution as a cornerstone of modern cloud infrastructure. By maintaining the foundational principles of its predecessors while adapting to the needs of a diverse and growing user base, Kubernetes has established itself as a pivotal force in shaping contemporary cloud and container management strategies.

## Evolution of Containers and the Rise of Kubernetes

### *Container Technology and Docker:*

The concept of containers, which are isolated environments for running applications, has been integral to Linux for many years. However, the introduction of Docker in 2013 was a game-changer, popularizing container technology in the software industry. Docker simplified the creation, deployment, and running of applications by allowing developers to package an application with its environment and dependencies into a single container unit. This innovation provided consistency across various development, testing, and production environments, and it paved the way for the widespread adoption of containers due to its portability, efficiency, and isolation capabilities.

### *Shift to Microservices:*

The adoption of containers coincided with a shift towards microservices architecture – a design approach where a large application is divided into small, independent services. Each service runs in its own container environment, making it lightweight and easily scalable. This modularity allows for quicker updates and better scalability compared to traditional, monolithic application architectures.

### *Need for Orchestration:*

The rapid proliferation of containerized microservices led to complexity in their management at scale. It became evident that a tool was needed to efficiently handle the lifecycle and interactions of numerous containers. This requirement led to the development of container orchestration tools, with Kubernetes emerging as the leader.

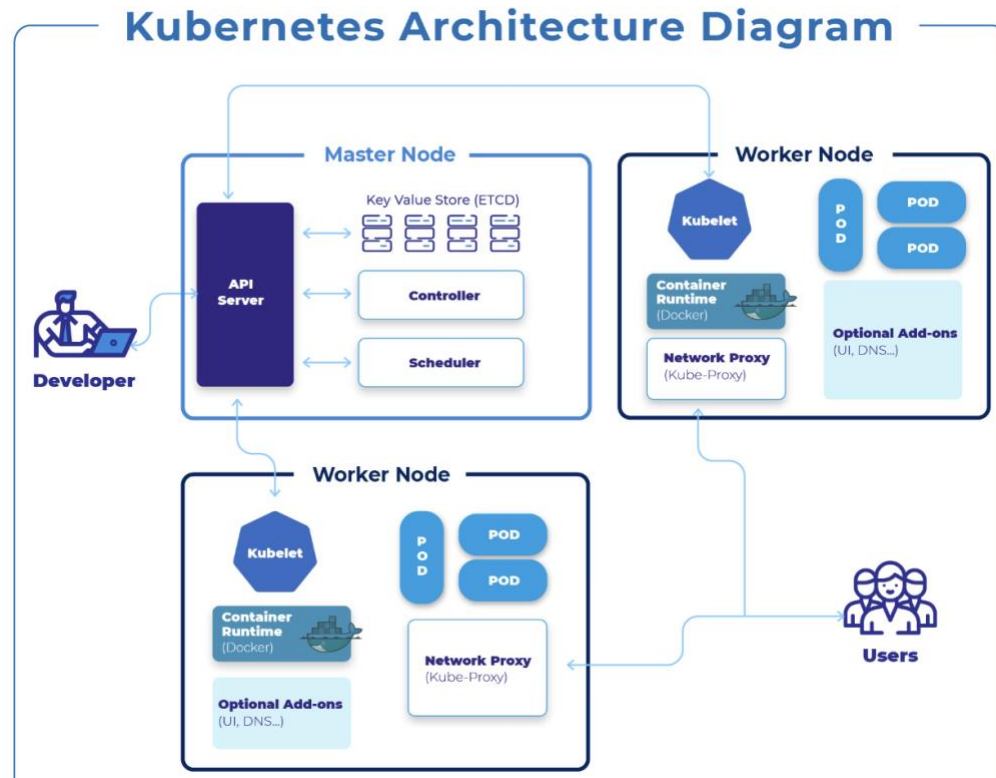
### *Kubernetes – A Solution to Container Orchestration:*

Originating from Google's experiences with Borg and Omega, Kubernetes was introduced in 2014 to manage containerized applications more effectively. It automated deployment, scaling, and operations of application containers across clusters of hosts, addressing key issues such as load balancing, auto-scaling, and self-healing. Kubernetes quickly became synonymous with container

orchestration, providing a robust, extensible way to manage containerized services at scale, further fueling the adoption of microservices and cloud-native technologies.

The evolution from Docker's simplification of container deployment to Kubernetes' sophisticated orchestration illustrates a fundamental shift in how applications are built and deployed. This synergy is a cornerstone in the modern cloud-native landscape, enabling businesses to deploy scalable, resilient applications efficiently and reliably.

## Kubernetes Architecture



### Overview of Core Components:

Kubernetes, an influential tool in modern software infrastructure, orchestrates containerized application management across a cluster of machines. Its architecture comprises several key components, each playing a crucial role in the system's functionality and efficiency.

- **Pods:** The basic building block of Kubernetes, a Pod represents a single instance of a running process in your cluster. Pods encapsulate one or several containers, their storage resources, a unique network IP, and options that govern their runtime behavior. Generally, Pods are transient and relatively ephemeral in nature.
- **Nodes:** A Node is a worker machine in Kubernetes, which can be either a physical or a virtual machine, depending on the cluster. Each Node is managed by the master, contains the services necessary to run Pods, and is monitored for status.
- **Services:** A Kubernetes Service is an abstraction layer which defines a logical set of Pods and a policy to access them. Services enable a loose coupling between dependent Pods,

providing a way to expose application or service functionality consistently and reliably within the Kubernetes network.

- **Deployment:** Managing Pods and ReplicaSets, a Deployment provides declarative updates to applications. It allows you to describe an application's life cycle, such as which images to use for the app, the number of Pods in a deployment, and the way to update them.

#### *Key Design Principles:*

**Declarative Configuration and Desired State Management:** Kubernetes is fundamentally built on a declarative model, where the user supplies the system with the desired state of their application, and Kubernetes works to maintain that state. This model abstracts the complexity involved in the deployment and scaling of applications, focusing on the "what" rather than the "how."

#### *Orchestration Mechanics in Kubernetes*

In addition to the core components and design principles, the essence of Kubernetes lies in its orchestration capabilities, which manage the lifecycle and operations of containers and applications across a distributed architecture. Understanding these mechanics further clarifies how Kubernetes simplifies complex container management.

- **Scheduling and Resource Allocation:** At the heart of Kubernetes orchestration is the scheduler, a component responsible for assigning work, in the form of Pods, to Nodes. It makes these decisions based on resource availability, constraints, and affinity specifications. This automated scheduling balances the load across the cluster, optimizing resource use and ensuring high availability.
- **Health Checks and Self-healing:** Kubernetes continuously monitors the state of Pods and Nodes. If a Pod fails due to software errors or underlying Node issues, the system automatically restarts or reschedules the Pod to maintain the desired state. This self-healing mechanism enhances the reliability and stability of applications.
- **Load Balancing and Service Discovery:** Kubernetes facilitates load balancing in two ways: internally within the cluster and externally for incoming traffic. Services in Kubernetes provide an abstract way to expose an application running on a set of Pods as a network service. With Kubernetes handling service discovery and routing, it evenly distributes network traffic so that the deployment is stable and efficient.
- **Scaling and Rollouts:** Kubernetes supports both horizontal and vertical scaling, which can be automated based on metrics like CPU usage, or manually managed. Deployments in Kubernetes enable rolling updates to applications, allowing you to update images, change configurations, or roll back to previous versions without downtime. This approach ensures continuous availability and dynamic adaptation to varying loads and conditions.
- **Configuration and Secret Management:** Managing and storing configuration and sensitive information is crucial. Kubernetes provides resources like ConfigMaps and Secrets to store non-confidential and confidential data separately. These can be used to manage environment-specific configurations and securely pass sensitive data to applications without hard-coding them into images.

- **Networking:** Kubernetes implements a flat network model wherein Pods can communicate with one another across Nodes. This model necessitates careful management of IP addresses, subnets, and port allocations. Network policies in Kubernetes allow the definition of how groups of Pods communicate with each other and other network endpoints.

#### *Integration with CI/CD*

Kubernetes's architecture and orchestration capabilities seamlessly integrate with continuous integration and continuous deployment (CI/CD) practices. It aligns with the DevOps philosophy, where automated pipelines can build, test, and deploy applications into Kubernetes environments, providing faster release cycles, greater scalability, and more robust applications.

### Kubernetes in MLOps

#### *Integrating Kubernetes in Machine Learning Operations (MLOps)*

Kubernetes has become a cornerstone in the realm of Machine Learning Operations (MLOps) by facilitating the management, deployment, and scalability of machine learning (ML) workflows. Its role in supporting MLOps stems from its ability to handle complex, distributed systems and workflows that are characteristic of modern ML and AI applications.

- **Managing ML Workflows:** Kubernetes efficiently orchestrates and manages the entire lifecycle of ML workflows. This includes data pre-processing, model training, evaluation, deployment, and monitoring. By using Kubernetes, teams can create reproducible ML pipelines which are crucial for consistent and reliable model development and deployment. Tools like Kubeflow, a Kubernetes-native open-source platform, extend Kubernetes capabilities to manage these workflows effectively.
- **Scalability and Repeatability:** ML workloads often require scaling compute resources up and down depending on the workload's nature, such as training large models or serving a high number of inference requests. Kubernetes excels in dynamically allocating resources based on demand, ensuring that ML workflows are not only scalable but also cost-efficient. Furthermore, Kubernetes' ability to replicate environments and processes guarantees repeatability, a critical factor in ML experiments and production deployments.



## Demo Objective

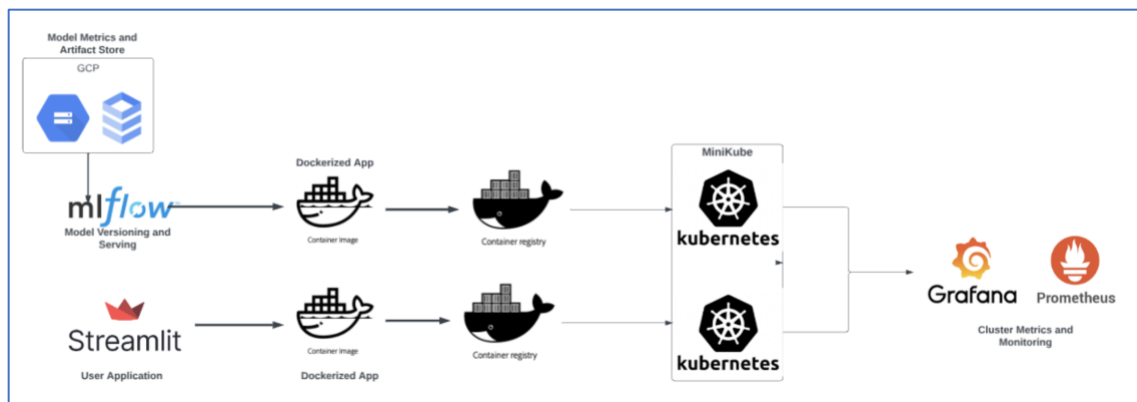
To effectively demonstrate deployment on Kubernetes Cluster I developed a MLOps pipeline to be deployed on Kubernetes Cluster. The same can be scaled to multiple data scientists working on a production grade system.

- **Step1:** Deploy a Machine learning model from Jupyter notebook into MLFlow Server as backend to track multiple experiments using MLFlow Experiments
  - MLFlow server requires a backend on PostgreSQL and GCP bucket to store artifacts. These need to be setup in advance to use MLFlow server
  - MLFlow server is deployed on Kubernetes
- **Step2:** Select a specific experiment we feel is good and deploy to production using MLFlow Server Models
- **Step3:** Access the deployed model via Streamlit Frontend (which is deployed on Kubernetes Cluster)
- **Step4:** Showcase the metrics of the cluster using Grafana and Prometheus. Grafana is a interactive dashboard and Prometheus is used for metrics measurement of the cluster. In this use case, I am monitoring Cluster Memory Usage, Cluster CPU Usage, Cluster Filesystem usage, Number of Pods and Input/Output bytes transferred in Cluster. All of these metrics are calculated in Prometheus and pulled into Grafana for visualization. Both services run on Kubernetes Cluster.

## Demo Setup

- System: Apple MacBook Pro 13.3 inch with 24GB Ram and 512GB Disk space
- Kubernetes Local deployment through Minikube with Docker driver
- Docker images created for
  - [MLFlow Containerization](#)
  - [Streamlit Containerization](#)
- Services Deployed on Kubernetes
  - MLFlow-tracking-server with backend via GCP PostgreSQL
  - Streamlit-App for frontend
  - Grafana – Monitoring and Alerting via Helm
  - Prometheus – Metric measurement via Helm

## Demo Architecture



## Demo Environment Setup Code

Commands to setup working environment for demo

- minikube start --memory 8192 cpus

```
~/inst2/streamlit$ minikube start --memory 8192 cpus 4
minikube v1.31.2 on Darwin 14.1 (arm64)
+ Automatically selected the docker driver
+ Using Docker Desktop driver with root privileges
+ Starting control plane node minikube in cluster minikube
+ Pulling base image ...
+ Creating docker container (CPUs=2, Memory=8192MB) ...
+ Preparing Kubernetes v1.27.4 on Docker 24.0.4 ...
  - Generating certificates and keys ...
  - Booting up control plane ...
  - Configuring RBAC rules ...
+ Configuring bridge CNI (Container Networking Interface) ...
+ Verifying Kubernetes components...
  - Using image gcr.io/k8s-minikube/storage-provisioner:v5
+ Enabled addons: storage-provisioner, default-storageclass
+ Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

- kubectl get svc

```
~/inst2/streamlit$ kubectl get svc
NAME         TYPE          CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP     10.96.0.1    <none>        443/TCP    68s
```

- Deploy mlflow-server

- **Step1:** Deploy configmap, secrets, mlflow deployment

```
~/inst2$ kubectl apply -f configmap_mlflow.yaml
configmap/mlflow-configmap created
~/inst2$ kubectl apply -f secrets_mlflow.yaml
secret/mlflow-postgresql-credentials created
~/inst2$ kubectl apply -f deployment_mlflow.yaml
deployment.apps/mlflow-tracking-server created
service/mlflow-tracking-server created
~/inst2$ kubectl get svc
NAME         TYPE          CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP     10.96.0.1    <none>        443/TCP    3m32s
mlflow-tracking-server  NodePort      10.102.116.104 <none>        5000:30001/TCP 5s
~/inst2$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
mlflow-tracking-server-869c59f575-pzz8v  0/1     ContainerCreating  0          8s
```

- **Step2:** port-forward mlflowserver

```
~/inst2$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
mlflow-tracking-server-869c59f575-pzz8v  1/1     Running    0          2m30s
~/inst2$ kubectl port-forward mlflow-tracking-server-869c59f575-pzz8v 5000
Forwarding from 127.0.0.1:5000 -> 5000
Forwarding from [::1]:5000 -> 5000
Handling connection for 5000
Handling connection for 5000
Handling connection for 5000
Handling connection for 5000
```

- Deploy Grafana in Kubernetes cluster via Helm

```
~/inst2$ helm install grafana grafana/grafana
NAME: grafana
LAST DEPLOYED: Wed Nov 1 11:20:17 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get your 'admin' user password by running:

  kubectl get secret --namespace default grafana -o jsonpath="{.data.admin-password}" | base64 --decode ; echo

2. The Grafana server can be accessed via port 80 on the following DNS name from within your cluster:

  grafana.default.svc.cluster.local

  Get the Grafana URL to visit by running these commands in the same shell:
  export POD_NAME=$(kubectl get pods --namespace default -l "app.kubernetes.io/name=grafana,app.kubernetes.io/instance=grafana" -o jsonpath="{.items[0].metadata.name}")
  kubectl --namespace default port-forward $POD_NAME 3000

3. Login with the password from step 1 and the username: admin

***** WARNING: Persistence is disabled!!! You will lose your data when *****
***** the Grafana pod is terminated. *****
~/inst2$ kubectl get svc
NAME         TYPE          CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
grafana      ClusterIP     10.96.0.172   <none>        80/TCP     6s
kubernetes   ClusterIP     10.96.0.1    <none>        443/TCP    10m
mlflow-tracking-server  NodePort      10.102.116.104 <none>        5000:30001/TCP 6m38s
~/inst2$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
grafana-75b5d88d49-8gcgl  1/1     Running    0          20s
mlflow-tracking-server-869c59f575-pzz8v  1/1     Running    0          6m52s
~/inst2$ kubectl port-forward grafana-75b5d88d49-8gcgl 3000
Forwarding from 127.0.0.1:3000 -> 3000
```

- Deploy Prometheus in Kubernetes cluster via Helm

```

helm install prometheus prometheus-community/prometheus
NAME: prometheus
LAST DEPLOYED: Wed Nov 1 11:23:28 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The Prometheus server can be accessed via port 80 on the following DNS name from within your cluster:
prometheus-server.default.svc.cluster.local

Get the Prometheus server URL by running these commands in the same shell:
export POD_NAME=$(kubectl get pods --namespace default -l "app.kubernetes.io/name=prometheus,app.kubernetes.io/instance=prometheus" -o jsonpath="{.items[0].metadata.name}")
kubectl --namespace default port-forward $POD_NAME 9090

The Prometheus alertmanager can be accessed via port 9093 on the following DNS name from within your cluster:
prometheus-alertmanager.default.svc.cluster.local

Get the Alertmanager URL by running these commands in the same shell:
export POD_NAME=$(kubectl get pods --namespace default -l "app.kubernetes.io/name=alertmanager,app.kubernetes.io/instance=prometheus" -o jsonpath="{.items[0].metadata.name}")
kubectl --namespace default port-forward $POD_NAME 9093

##### WARNING: Pod Security Policy has been disabled by default since #####
##### it deprecated after k8s 1.25+, use #####
##### (index .Values "prometheus-node-exporter" "rbac" #####
##### "pspEnabled") with (index .Values #####
##### "prometheus-node-exporter" "rbac" "pspAnnotations") #####
##### in case you still need it. #####

The Prometheus PushGateway can be accessed via port 9091 on the following DNS name from within your cluster:
prometheus-prometheus-pushgateway.default.svc.cluster.local

Get the PushGateway URL by running these commands in the same shell:
export POD_NAME=$(kubectl get pods --namespace default -l "app=prometheus-pushgateway,component=pushgateway" -o jsonpath="{.items[0].metadata.name}")
kubectl --namespace default port-forward $POD_NAME 9091

For more information on running Prometheus, visit:
https://prometheus.io/

kubectl get svc
NAME                                TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
grafana                             ClusterIP     10.96.217.172  <none>         80/TCP           4m46s
kubernetes                           ClusterIP     10.96.0.1      <none>         443/TCP          14m
mflow-tracking-server               NodePort      10.102.116.104 <none>         5000:30001/TCP   11m
prometheus-alertmanager             ClusterIP     10.107.99.133  <none>         9093/TCP         95s
prometheus-alertmanager-headless    ClusterIP     None           <none>         9093/TCP         95s
prometheus-kube-state-metrics        ClusterIP     10.104.162.151 <none>         8080/TCP         95s
prometheus-prometheus-node-exporter  ClusterIP     10.111.51.23   <none>         9100/TCP         95s
prometheus-prometheus-pushgateway    ClusterIP     10.98.200.63   <none>         9091/TCP         95s
prometheus-server                   ClusterIP     10.96.149.74   <none>         80/TCP           95s

kubectl get pods
NAME                                READY    STATUS    RESTARTS    AGE
grafana-75b5d8d49-8gcgl             1/1     Running   0            4m50s
mflow-tracking-server-869c59f575-pzz8v 1/1     Running   0            11m
prometheus-alertmanager-0            1/1     Running   0            99s
prometheus-kube-state-metrics-5b74ccb6b4-z8bcs 1/1     Running   0            99s
prometheus-prometheus-node-exporter-sgrff 1/1     Running   0            99s
prometheus-prometheus-pushgateway-79ff799669-b5xw6 1/1     Running   0            99s
prometheus-server-5bff8cd5d-m856v      2/2     Running   0            99s

```

- Deploy streamlit app into Kubernetes cluster

```

~/inst2/streamlit kubectl apply -f deployment_streamlit.yaml
deployment.apps/streamlit-app unchanged
service/streamlit-app-service created

~/inst2/streamlit kubectl get svc
NAME                                TYPE          CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
grafana                             ClusterIP     10.96.217.172  <none>         80/TCP           10m
kubernetes                           ClusterIP     10.96.0.1      <none>         443/TCP          19m
mflow-tracking-server               NodePort      10.102.116.104 <none>         5000:30001/TCP   16m
prometheus-alertmanager             ClusterIP     10.107.99.133  <none>         9093/TCP         6m49s
prometheus-alertmanager-headless    ClusterIP     None           <none>         9093/TCP         6m49s
prometheus-kube-state-metrics        ClusterIP     10.104.162.151 <none>         8080/TCP         6m49s
prometheus-prometheus-node-exporter  ClusterIP     10.111.51.23   <none>         9100/TCP         6m49s
prometheus-prometheus-pushgateway    ClusterIP     10.98.200.63   <none>         9091/TCP         6m49s
prometheus-server                   ClusterIP     10.96.149.74   <none>         80/TCP           6m49s
streamlit-app-service               NodePort      10.96.224.121  <none>         80:31344/TCP     13s

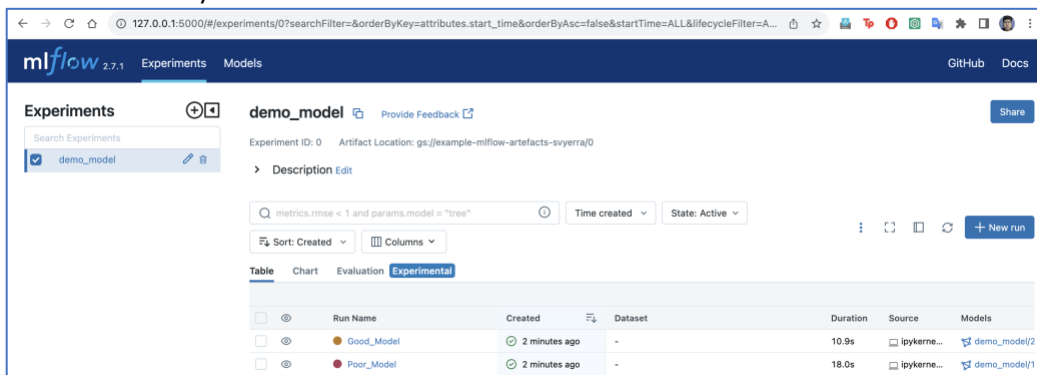
~/inst2/streamlit kubectl get pods
NAME                                READY    STATUS    RESTARTS    AGE
grafana-75b5d8d49-8gcgl             1/1     Running   0            10m
mflow-tracking-server-869c59f575-pzz8v 1/1     Running   0            16m
prometheus-alertmanager-0            1/1     Running   0            6m52s
prometheus-kube-state-metrics-5b74ccb6b4-z8bcs 1/1     Running   0            6m52s
prometheus-prometheus-node-exporter-sgrff 1/1     Running   0            6m52s
prometheus-prometheus-pushgateway-79ff799669-b5xw6 1/1     Running   0            6m52s
prometheus-server-5bff8cd5d-m856v      2/2     Running   0            6m52s
streamlit-app-6d6f74d9c8-b24zf        1/1     Running   0            2m24s
streamlit-app-6d6f74d9c8-qlbgw        1/1     Running   0            2m24s

~/inst2/streamlit kubectl port-forward streamlit-app-6d6f74d9c8-b24zf 8501
Forwarding from 127.0.0.1:8501 -> 8501

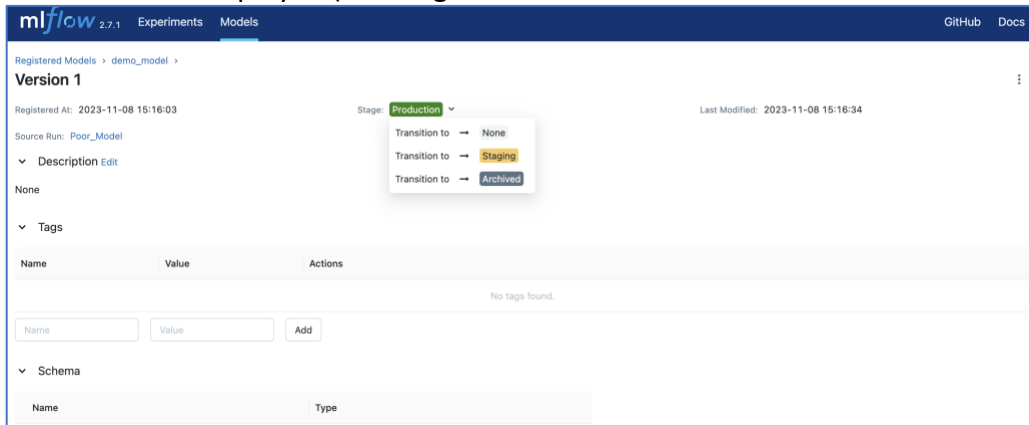
```

## Demo Deployment Screenshots

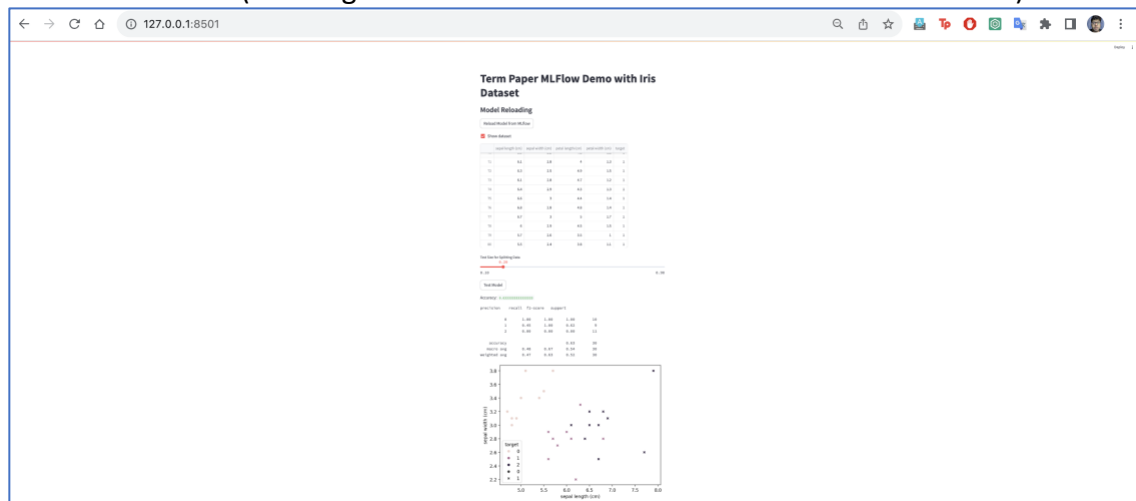
- MLFlow logging Experiments (Running on Kubernetes Cluster – Localhost: 127.0.0.1:5000)



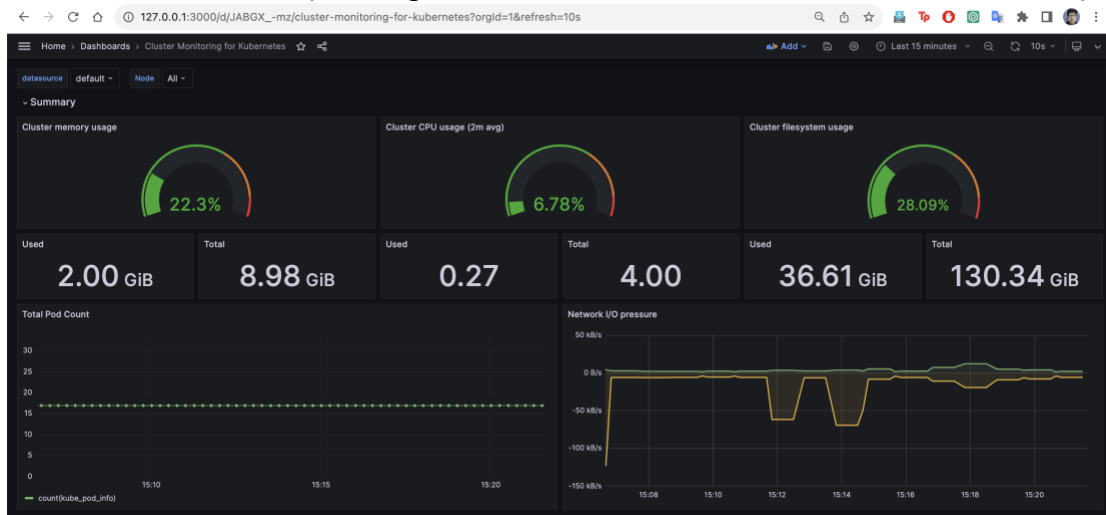
- MLFlow model deployed (Running on Kubernetes Cluster – Localhost: 127.0.0.1:5000)



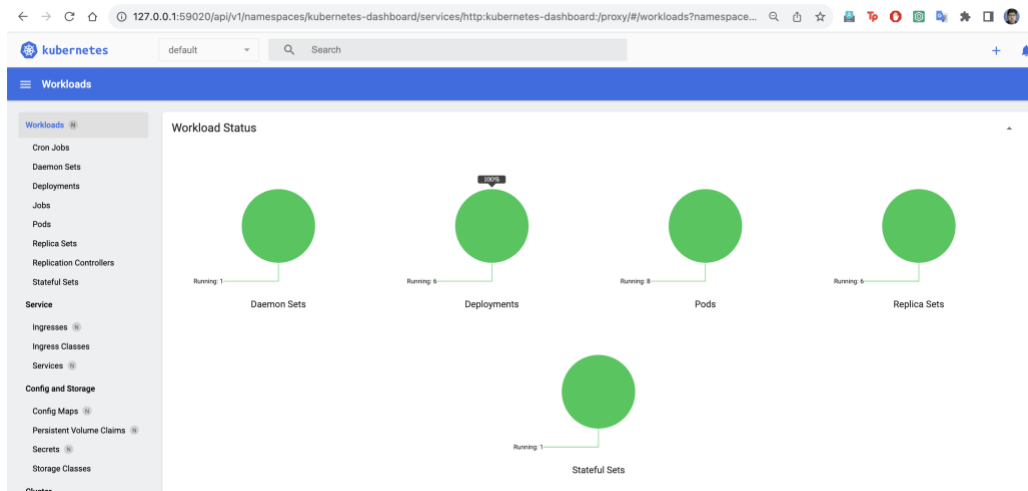
- Streamlit Frontend (Running on Kubernetes Cluster – Localhost: 127.0.0.1:8501)



- Grafana + Prometheus (Running on Kubernetes Cluster – Localhost: 127.0.0.1:3000)



- Cluster Details via Minikube Dashboard

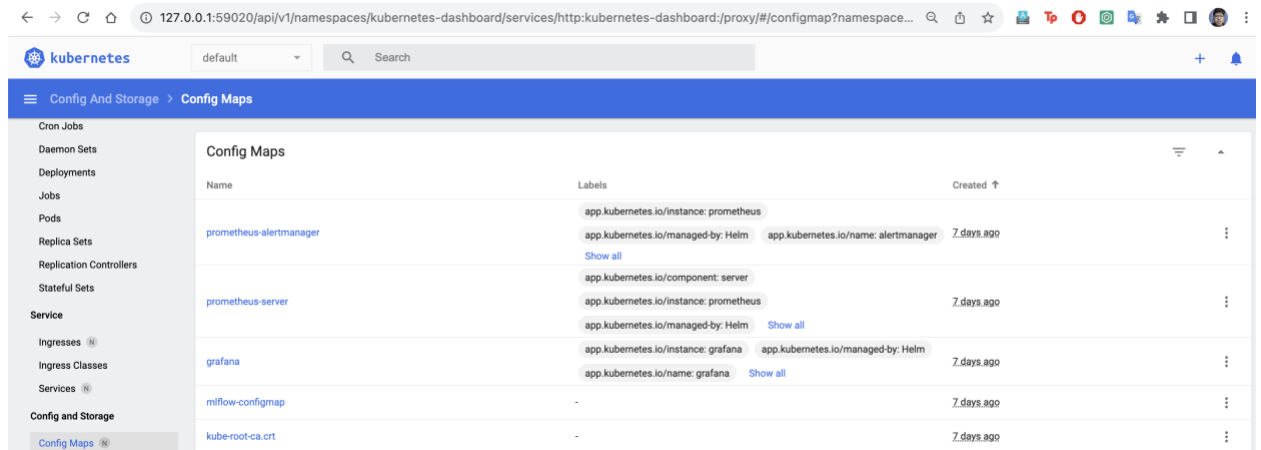


- Cluster Deployments

The screenshot shows the Minikube Dashboard with the 'Deployments' section selected. The 'Deployments' table lists the following deployments:

Name	Images	Labels	Pods	Created
streamlit-app	sandeepyerra/streamlit-app:latest	app.kubernetes.io/component: metrics	1 / 1	7 days ago
prometheus-kube-state-metrics	registry.k8s.io/kube-state-metrics/kube-state-metrics-v2.10.0	app.kubernetes.io/instance: prometheus app.kubernetes.io/managed-by: Helm	1 / 1	7 days ago
prometheus-prometheus-pushgateway	quay.io/prometheus/pushgateway:v1.6.0	app.kubernetes.io/instance: prometheus app.kubernetes.io/managed-by: Helm app.kubernetes.io/name: prometheus-pushgateway	1 / 1	7 days ago
prometheus-server	quay.io/prometheus-operator/prometheus-config-reloader:v0.67.0 quay.io/prometheus/prometheus:v2.47.0	app.kubernetes.io/component: server app.kubernetes.io/instance: prometheus app.kubernetes.io/managed-by: Helm	1 / 1	7 days ago
grafana	docker.io/grafana/grafana:10.1.5	app.kubernetes.io/instance: grafana app.kubernetes.io/managed-by: Helm app.kubernetes.io/name: grafana	1 / 1	7 days ago
mlflow-tracking-server	sandeepyerra/mlflow-image:v2.1	app: mlflow-tracking-server	1 / 1	7 days ago

- Cluster ConfigMaps



The screenshot shows the Kubernetes dashboard interface. The left sidebar contains a navigation menu with categories like Cron Jobs, Daemon Sets, Deployments, Jobs, Pods, Replica Sets, Replication Controllers, Stateful Sets, Service, Ingresses, Ingress Classes, Services, and Config and Storage. The 'Config and Storage' section is expanded, showing 'Config Maps'. The main panel displays a table of Config Maps in the 'default' namespace.

Name	Labels	Created
<a href="#">prometheus-alertmanager</a>	<code>app.kubernetes.io/instance: prometheus</code> <code>app.kubernetes.io/managed-by: Helm</code> <code>app.kubernetes.io/name: alertmanager</code> <a href="#">Show all</a>	7 days ago
<a href="#">prometheus-server</a>	<code>app.kubernetes.io/component: server</code> <code>app.kubernetes.io/instance: prometheus</code> <code>app.kubernetes.io/managed-by: Helm</code> <a href="#">Show all</a>	7 days ago
<a href="#">grafana</a>	<code>app.kubernetes.io/instance: grafana</code> <code>app.kubernetes.io/managed-by: Helm</code> <code>app.kubernetes.io/name: grafana</code> <a href="#">Show all</a>	7 days ago
<a href="#">miflow-configmap</a>	-	7 days ago
<a href="#">kube-root-ca.crt</a>	-	7 days ago