

# DEPLOYMENT OF WIRELESS AND LOW LATENCY EDGE CLUSTER

*A report submitted in partial fulfilment of the requirements*

*for the award of the degree of*

*B.Tech Computer Science and Engineering*

*by*

P Lakshmi Sujitha (Roll No: 120CS0017)

Bhavishya Yekollu (Roll No: 120CS0025)

K. R. Ramya Shri Shakthi (Roll No: 120CS0046)

Satakshi Sinha (Roll No: 120CS0047)

Under the Guidance of

Dr. Anil Kumar R

Assistant Professor

Dept. of CSE, IIITDM Kurnool



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING  
INDIAN INSTITUTE OF INFORMATION TECHNOLOGY DESIGN  
AND MANUFACTURING KURNOOL

November 2023

# Evaluation Sheet

**Title of the Project:**

**Name of the Student(s):**

**Examiner(s):**

-----  
-----

**Supervisor(s):**

-----  
-----

**Head of the Department:**

-----

**Date:**

**Place:**

# Certificate

We, **P. Lakshmi Sujitha, Bhavishya Yekollu, K. R. Ramya Shri Shakthi and Satakshi Sinha**, with Roll numbers: **120CS0017, 120CS0025, 120CS0046 and 120CS0047** respectively, hereby declare that the material presented in the Project Report titled **Project Title** represents original work carried out by us in the **Department of Computer Science and Engineering** at the **Indian Institute of Information Technology Design and Manufacturing Kurnool** during the years **2023 - 2024**. With our signatures, we certify that:

- We have not manipulated any of the data or results.
- We have not committed any plagiarism of intellectual property. We have clearly indicated and referenced the contributions of others.
- We have explicitly acknowledged all collaborative research and discussions.
- We have understood that any false claim will result in severe disciplinary action.
- We have understood that the work may be screened for any form of academic misconduct.

Date:

Student's Signature

In my capacity as supervisor of the above-mentioned work, I certify that the work presented in this Report is carried out under my supervision, and is worthy of consideration for the requirements of B.Tech. Project work.

Advisor's Name:

Advisor's Signature

# *Abstract*

This project signifies the successful realization of wireless, event-driven, and low-latency edge cluster. Employing cutting-edge technologies such as Kubernetes, Docker, Grafana, Prometheus, Node.js, MongoDB and Ceph, we've crafted a versatile platform applicable across diverse domains. Kubernetes orchestrates efficiently, Docker enables seamless deployment and Grafana with Prometheus ensures robust monitoring, enhancing fault tolerance. Leveraging Node.js with MongoDB, we've developed responsive, event-driven applications. Although low-latency implementation is partial, the groundwork is set for future optimizations, promising even greater efficiency. The edge cluster addresses challenges in Kubernetes environments, showcasing automatic deployment and efficient vertical scaling, adapting dynamically to varying workloads.

## *Acknowledgements*

We would extend our sincerest gratitude to Dr. Anil Kumar R, our mentor, for his invaluable guidance and support throughout this project. We would also like to express gratitude to Dr. K. Sathya Babu, the Head of the Department, the entire Computer Science and Engineering (CSE) department, and our esteemed institute for providing us with this opportunity.

# Contents

<b>Evaluation Sheet</b>	<b>i</b>
<b>Certificate</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Acknowledgements</b>	<b>iv</b>
<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>vii</b>
<b>Abbreviations</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Challenges . . . . .	2
1.3 Objectives of Minor project . . . . .	3
<b>2 Technologies used</b>	<b>4</b>
2.1 Docker . . . . .	4
2.2 Kubernetes . . . . .	5
2.2.1 Kubernetes Architecture: . . . . .	5
2.3 Prometheus . . . . .	7
2.4 Grafana . . . . .	8
2.5 MongoDB . . . . .	8
2.6 Node.js . . . . .	8
2.7 Express . . . . .	9
<b>3 System design and Implementation</b>	<b>10</b>
3.1 Kubernetes cluster setup . . . . .	11
3.1.1 Microk8s: . . . . .	11

---

3.1.2	Minikube:	12
3.1.3	Kubeadm	13
3.2	Efficient Deployment	14
3.3	Automated vertical scaling	15
3.4	Distributed storage	15
3.5	Low latency and event driven cluster	16
3.5.1	Implementation:	17
3.6	Application	18
<b>4</b>	<b>Testing and Results</b>	<b>20</b>
4.1	Use Cases that can benefit by using our edge cluster	20
4.2	Testing and Results	21
4.2.1	Testing the application	21
4.2.2	Results of multiple features of our cluster	21
<b>5</b>	<b>Summary and Future Directions</b>	<b>24</b>
5.1	Conclusion	24
5.2	Future Works	25
<b>6</b>	<b>References</b>	<b>26</b>

# List of Figures

2.1	Docker Architecture . . . . .	4
2.2	Kubernetes Architecture . . . . .	6
3.1	Master and worker Architecturre . . . . .	10
3.2	GUI when heart rate is normal . . . . .	18
3.3	GUI when heart rate is abnormal . . . . .	19
4.1	Kubernetes Setup . . . . .	21
4.2	Prometheus and grafana pods running in the cluster . . . . .	21
4.3	Applications we have tried on our cluster . . . . .	22
4.4	Pods that are running in rook ceph . . . . .	22
4.5	Pods that are deployed once the kubernetes is created. . . . .	22
4.6	Untainting low latency node to deploy the application . . . . .	23



# Abbreviations

<b>K8S</b>	<b>K</b> ubernetes
<b>I/O</b>	<b>I</b> ntput/ <b>O</b> utput
<b>VM</b>	<b>V</b> irtual <b>M</b> achine
<b>IoT</b>	<b>I</b> nternet <b>o</b> f <b>T</b> hings
<b>GUI</b>	<b>G</b> raphical <b>U</b> ser <b>I</b> nterface
<b>JSON</b>	<b>J</b> ava <b>S</b> cript <b>O</b> bject <b>N</b> otation
<b>SQL</b>	<b>S</b> tructured <b>Q</b> uery <b>L</b> anguage
<b>CRI</b>	<b>C</b> ontainer <b>R</b> untime <b>I</b> nterface
<b>CNI</b>	<b>C</b> ontainer <b>N</b> etwork <b>I</b> nterface
<b>HTTP</b>	<b>H</b> yper <b>T</b> ext <b>T</b> ransfer <b>P</b> rotocol
<b>REST</b>	<b>R</b> epresentational <b>S</b> tate <b>T</b> ransfer
<b>API</b>	<b>A</b> pplication <b>P</b> rogramming <b>I</b> nterface
<b>CI/CD</b>	<b>C</b> ontinuous <b>I</b> ntegration and <b>C</b> ontinuous <b>D</b> eployment

*Dedicated to IIITDM Kurnool...*

# Chapter 1

## Introduction

Over the last few years, the popularity of edge computing has been increasing, driven by the growing adoption of IoT devices, the need for low-latency applications, and advancements in technologies like 5G. Industries such as manufacturing, healthcare, transportation, agriculture and smart cities are harnessing the power of edge computing to boost operational efficiency and responsiveness.

An edge cluster refers to a distributed computing infrastructure that brings computation and data storage closer to the source of data generation. In traditional cloud computing, data is sent to a centralized cloud server for processing and analysis. In contrast, edge computing processes data locally on devices or on servers located closer to the data source, offering several key features and advantages over traditional cloud computing.

One of its primary benefits is low latency, achieved by reducing the distance the data travels between devices and centralized servers. This is crucial for applications requiring real-time processing, like IoT devices and autonomous vehicles. Edge computing enhances bandwidth efficiency by locally processing and transmitting only relevant data, reducing the load on network resources. Additionally, it prioritizes privacy and security by keeping sensitive information localized, minimizing the risk of data breaches. The scalability of edge computing allows for the efficient distribution of computing resources, catering to large and variable workloads. Furthermore, it supports offline operation, ensuring devices can function independently of cloud connectivity. Real-time decision-making is another advantage, as edge computing enables on-the-spot data processing, particularly valuable in time-sensitive and critical applications.

In this project, we are deploying a wireless edge cluster using many tools like Kubernetes, Docker, Prometheus and Grafana. Our primary goal is to minimize the latency for the users while making the cluster fault tolerant. We are utilizing our personal computers to deploy this setup.

## 1.1 Motivation

Our project aims to establish an edge cluster architecture characterized by fault tolerance, scalability, and minimal latency. We are using kubeadm, a Kubernetes tool, to set up and manage the edge cluster which offers many features like horizontal scaling, services for work load balancing, persistent volumes to prevent data loss, etc. However, kubeadm doesn't inherently support automatic deployment, i.e, automatic addition of worker nodes to the cluster. Our cluster implementation offers a solution for achieving automatic addition of nodes to an extent, making the joining process efficient and simple. Our edge cluster also aims to minimize the latency so that the users have a seamless, interactive user experience while also improving user productivity especially in scenarios where time sensitive actions are involved.

## 1.2 Challenges

Deploying an edge clusters using kubeadm, we faced some unique challenges due to the distributed and resource-constrained nature of edge environment which include:

1. **Resource limitations:** Our edge cluster uses our laptops as edge nodes, the resources like CPU, memory, storage were limited, which affected the performance of the cluster.
2. **Network Connectivity:** Since our edge cluster uses a private network, the network connectivity is intermittent, unreliable and unstable.
3. **Hardware Diversity:** Our edge nodes have different hardware architectures, such as x86, ARM etc. We were still able to ensure compatibility.
4. **Network Topologies:** Our edge cluster can involve complex network topologies, including scenarios where nodes are geographically dispersed. Our cluster can still function, as long as all the nodes are in the same network.

5. **Deployment Automation:** Setting up and managing clusters at the edge manually is impractical. We have automated this process to an extent to make it more efficient and simple.

6. **Node Lifecycle Management:** Our edge cluster experiences dynamic changes in node availability due to factors like devices being powered on/off.

7. **Application Placement:** Deciding where to deploy applications within the edge cluster to optimize for latency and resource usage can be challenging. We have tried to solve this issue considering multiple factors.

8. **Monitoring and Debugging:** Monitoring and debugging edge cluster was challenging due to limited visibility into remote nodes. We have used some tools like Grafana and Prometheus to solve this challenges.

### 1.3 Objectives of Minor project

Our project aims to deploy an edge cluster that effectively addresses the challenges mentioned above. The deployment is designed to optimize resource usage and ensure high availability, ultimately minimizing latency and providing an efficient and secure environment for applications. Our project also has an objective of minimizing latency in our edge cluster deployment by strategically placing applications on nodes in close proximity to the master node. This optimization ensures faster response times, as the applications leverage the reduced communication distance, resulting in an enhanced and efficient user experience.

## Chapter 2

# Technologies used

### 2.1 Docker

Docker serves as a platform enabling developers to create, package, and deploy applications as containers. These containers are self-contained and lightweight packages encompassing everything necessary to run an application, from code to libraries and dependencies. This standard packaging approach simplifies application migration across environments and diverse platforms.

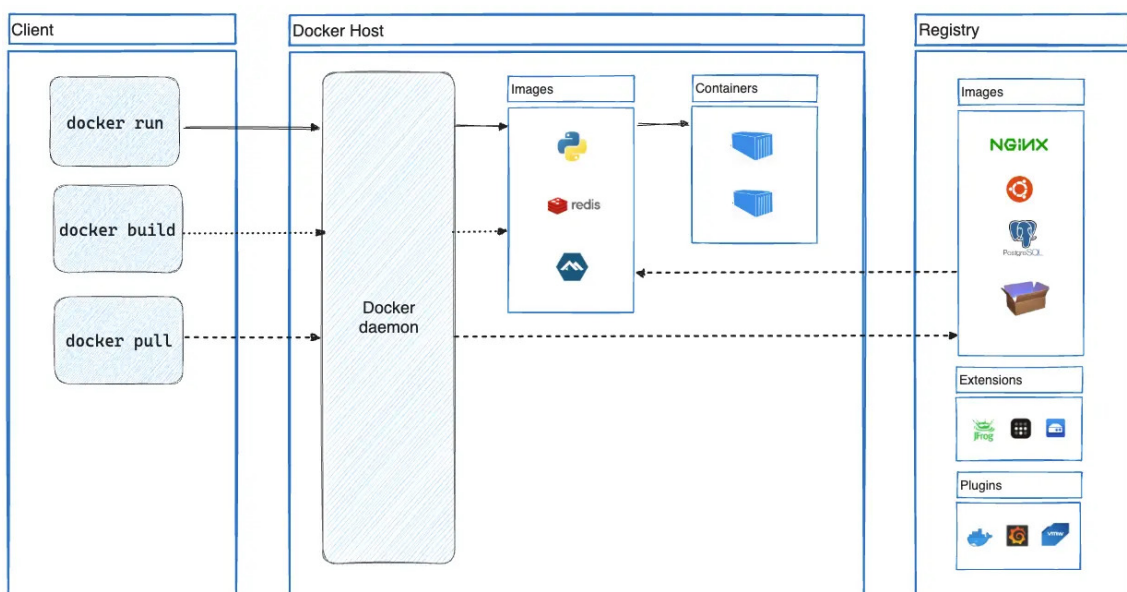


FIGURE 2.1: Docker Architecture

At the heart of Docker lies the Docker Engine, a portable runtime that facilitates container execution across multiple operating systems. This Engine comprises essential components:

- 1. Docker daemon:** Responsible for managing Docker entities such as images, containers, and networks on the server-side.
- 2. Docker client:** A command-line interface (CLI) tool facilitating user interaction with Docker entities by communicating with the Docker daemon.
- 3. Docker API:** An interface allowing developers to programmatically access, control, and supervise Docker entities via code.

Docker Engine functions through a client-server model: commands from the Docker client are executed by the Docker daemon. The Docker API employs a RESTful interface, enabling developers to communicate with Docker objects through HTTP requests.

Overall, Docker stands as a robust platform empowering the creation and deployment of containerized applications, with the Docker Engine serving as the pivotal component driving this capability.

## 2.2 Kubernetes

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Kuberenetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.

### 2.2.1 Kubernetes Architecture:

A K8S (Kubernetes) cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

**1. Pod:** Pod is a single instance of an application which is also the smallest unit of execution in Kubernetes. A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context.

**2. Node:** Kubernetes runs your workload by placing containers into Pods to run on Nodes. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the control plane and contains the services necessary to run Pods.

The components on a node include the kubelet, a container runtime, and the kube-proxy:

- **Kubelet:** The Kubelet is responsible for managing the deployment of pods to Kubernetes nodes. It receives commands from the API server and instructs the container runtime to start or stop containers as needed.
- **Container runtime:** A container runtime, also known as container engine, is a software component that can run containers on a host operating system.
- **Container Runtime Interface:** The Container Runtime Interface (CRI) is the main protocol for the communication between the kubelet and Container Runtime.
- **Kube-proxy:** kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. kube-proxy maintains network

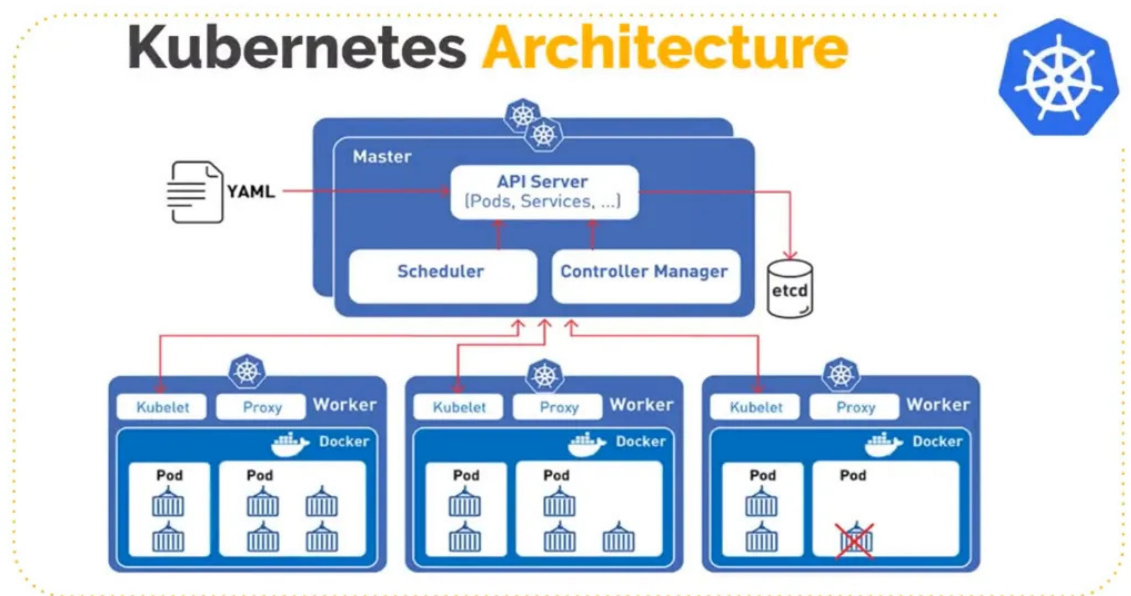


FIGURE 2.2: Kubernetes Architecture



rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

**3. Control Plane:** The Kubernetes control plane manages clusters and resources such as worker nodes and pods. The control plane receives information such as cluster activity, internal and external requests, and more. Based on these factors, the control plane moves the cluster resources from their current state to the desired state.

Control Plane components: The control plane's components make global decisions about the cluster, as well as detecting and responding to cluster events.

- **kube-apiserver:** The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.
- **etcd:** Consistent and highly-available key value store used as Kubernetes backing store for all cluster data.
- **kube-scheduler:** Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

**4. Kube Controller Manager:** It manages various controllers in Kubernetes. Controllers are control loops that continuously watch the state of your cluster, then make or request changes where needed. Controllers continuously talk to the kube-apiserver and the kube-apiserver receives all information of nodes through Kubelet.

## 2.3 Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit, collects and stores its metrics as time series data, i.e. metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels. Prometheus's main features are:

- a multi-dimensional data model with time series data identified by metric name and key/value pairs
- PromQL, a flexible query language to leverage this dimensionality

- no reliance on distributed storage; single server nodes are autonomous
- time series collection happens via a pull model over HTTP
- Pushing time series is supported via an intermediary gateway
- targets are discovered via service discovery or static configuration
- multiple modes of graphing and dashboarding support

## 2.4 Grafana

Grafana open source software enables you to query, visualize, alert on, and explore your metrics, logs, and traces wherever they are stored. Grafana OSS provides you with tools to turn your time-series database data into insightful graphs and visualizations. The Grafana OSS plugin framework also enables you to connect other data sources like NoSQL/SQL databases, ticketing tools like Jira or ServiceNow, and CI/CD tooling like GitLab.

## 2.5 MongoDB

MongoDB is a document-based NoSQL database that is used to store and retrieve data in our cloud service. It is highly scalable and provides fast read and write performance, which makes it an ideal choice for building cloud applications.

MongoDB stores data in JSON-like documents, which makes it easy to work with data in a natural and intuitive way. There is no need to define a fixed schema for the data. MongoDB can handle petabytes of data, and it can scale horizontally by adding more nodes to the cluster. This makes it easy to scale the database as our application grows, and it provides high availability and fault tolerance by distributing the data across multiple nodes.

## 2.6 Node.js

Node.js is a powerful runtime environment for building server-side applications. It is built on top of the V8 JavaScript engine, which is the same engine used by Google Chrome.

This means that Node.js can run JavaScript code on the server, which makes it easy to write code that can run on both the frontend and the backend.

Node.js provides a non-blocking I/O model, which allows our application to handle multiple requests simultaneously without blocking the event loop. This makes it easy to build scalable and efficient applications that can handle a large number of concurrent connections.

## 2.7 Express

Express is a minimalist web framework for Node.js that provides a simple and flexible API for building web applications. It is widely used in the industry and is known for its scalability and ease of use.

One of the key features of Express is its ability to handle HTTP requests and responses. Express provides a number of methods that allow developers to define routes, handle requests, and send responses to clients. This makes it easy to build RESTful APIs and web applications.

Another important feature of Express is its middleware architecture. Middleware functions are functions that have access to the request and response objects, and can perform tasks such as logging, authentication, and data parsing. Express provides a number of built-in middleware functions, and it is also easy to create custom middleware functions.

## Chapter 3

# System design and Implementation

In a Kubernetes cluster, there are two main types of nodes: master nodes and worker nodes. Each type of node has distinct roles and responsibilities.

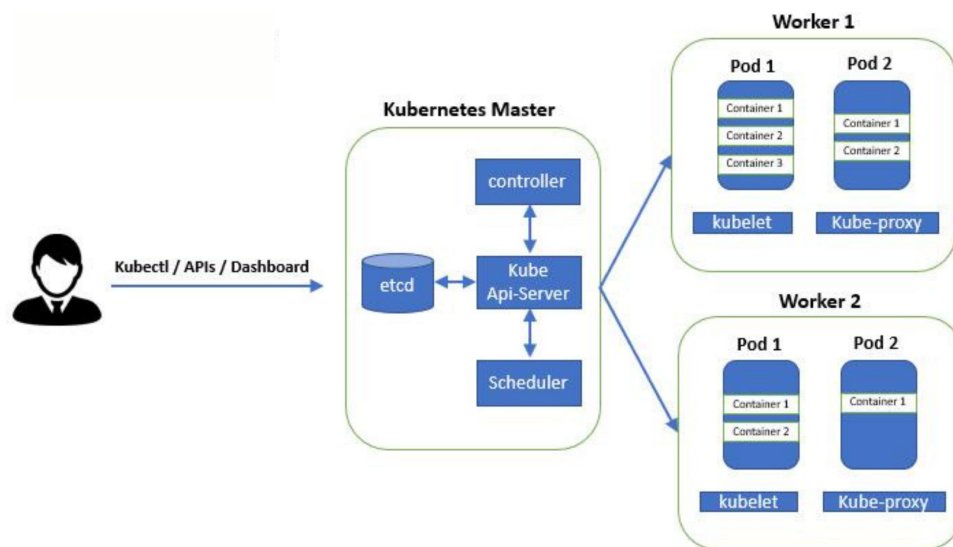


FIGURE 3.1: Master and worker Architecture

### 1. Master Nodes:

The master nodes form the control plane of the Kubernetes cluster and are responsible for managing the overall state of the cluster. The key components running on the master

nodes include the API server, which acts as the entry point for all administrative tasks and communication with the cluster; the etcd data store, which stores the configuration data and the current state of the cluster; the controller manager, which oversees the desired state of the cluster and ensures that it matches the actual state; and the scheduler, which assigns workloads (containers) to worker nodes based on resource availability and constraints. The master nodes play a crucial role in coordinating and orchestrating the deployment, scaling, and management of applications across the cluster. They are also responsible for handling events, monitoring cluster health, and responding to changes in the cluster's state.

**2. Worker Nodes:** Worker nodes, also known as minion nodes, are responsible for running the actual application workloads in the form of containers. These nodes host the deployed containers and provide the necessary runtime environment. Each worker node runs a container runtime (such as Docker or containerd) and the Kubernetes agent, known as Kubelet. The Kubelet is responsible for communicating with the master node, ensuring that containers are running as expected, and reporting the node's status. Additionally, worker nodes may run a container networking interface to facilitate communication between containers across different nodes and a container storage interface for managing storage resources. Worker nodes collectively contribute computing resources to the cluster, and their primary role is to execute and manage the containers that make up the applications deployed on the cluster. The combination of master and worker nodes in a Kubernetes cluster forms a distributed and scalable platform for containerized application orchestration and management.

## 3.1 Kubernetes cluster setup

Kubernetes is a powerful container orchestration platform, and it is possible to set up a cluster using various tools like kubeadm, microk8s, and minikube. Each tool has its own advantages and disadvantages:

### 3.1.1 Microk8s:

#### Advantages:

- **Single-Node and Multi-Node Clusters:** MicroK8s can be configured to run both single-node and multi-node clusters, providing flexibility for different use cases.

- **Snap Package:** MicroK8s is distributed as a snap package, making installation and updates straightforward on Ubuntu systems.
- **Add-Ons:** MicroK8s comes with built-in add-ons, such as storage, DNS, and dashboard, making it easy to enable additional features.

**Disadvantages:**

- **Ubuntu Focused:** While it can be used on other Linux distributions, MicroK8s is initially designed with a focus on Ubuntu systems.
- **Resource Usage:** Similar to Minikube, running MicroK8s on resource-constrained machines may still impact performance.

**3.1.2 Minikube:****Advantages:**

- **Ease of Use:** Minikube is designed to be a simple and lightweight solution for running a single-node Kubernetes cluster on a local machine. It's easy to install and get started.
- **Local Development:** Ideal for developers who want to test and develop Kubernetes applications on their local machines before deploying to a larger cluster.
- **Hypervisor Agnostic:** Minikube supports multiple hypervisors, including VirtualBox, KVM, Hyper-V, and Docker, providing flexibility for different environments.

**Disadvantages:**

- **Single-Node Cluster:** Minikube is primarily designed for local development and testing, and it runs a single-node Kubernetes cluster. It is not suitable for scenarios that require multi-node clusters.
- **Resource Intensive:** While it's lightweight for a local cluster, running Minikube with a full-fledged Kubernetes environment may still consume a significant amount of system resources.

### 3.1.3 Kubeadm

#### Advantages:

- **Flexibility and Customization:** Kubeadm provides a basic approach to set up a Kubernetes cluster, allowing for greater customization and adaptability according to specific requirements.
- **Production-Ready:** Kubeadm is designed with production use in mind. It is suitable for creating clusters on servers, making it a good choice for production environments.
- **Cluster Expansion:** Kubeadm supports cluster expansion, allowing the addition of new nodes to an existing cluster.

In our pursuit of creating a Kubernetes cluster for testing and learning purposes, we initially opted for Minikube, which efficiently sets up a single-node cluster. However, since we wanted multi-node configuration with one master node and three worker nodes (the nodes being our respective laptops), we transitioned to using Kubeadm to setup the cluster, which demanded lots of research and debugging.

Initially, we installed prerequisites on all the nodes:

- Ubuntu 22.04 as the operating system
- Docker
- Kubernetes tools such as kubeadm, kubelet, and kubectl

Recognizing the importance of a Container Runtime Interface (CRI), we then installed **CRI-O** for its stability, overcoming challenges encountered with other CRIs like Containerd.

The master node was then initialized using the command **kubeadm init --pod-network-cidr=10.244.0.0/16**. This command kickstarted the process of configuring the master node, setting up the Kubernetes control plane components, and generating tokens for worker node authentication.

Following the master node initialization, the next step involved installing a Container Network Interface (CNI) to facilitate communication between pods across the cluster. In

this case, we opted for **Flannel**, a popular CNI solution known for its simplicity and effectiveness in creating a virtual network for container communication.

The final step in the cluster setup process was joining the worker nodes to the master. This step involved using the tokens generated during the master node initialization process, allowing worker nodes to authenticate and join the cluster seamlessly.

## 3.2 Efficient Deployment

Kubernetes, being a powerful container orchestration system, provides robust capabilities for managing containerized applications but may require manual intervention during the deployment phase. In response to this limitation, our initiative involves streamlining and automating deployment steps to enhance efficiency.

To achieve this, we have devised a solution where all the essential deployment commands are consolidated into a file by a master node as soon as it is created. This file acts as a deployment script that, when executed, automates the joining process of worker nodes into the Kubernetes cluster. The orchestration begins with the master node broadcasting this deployment script to all worker nodes. Upon receiving the file, each worker node autonomously executes the script, facilitating its automatic integration into the cluster.

By encapsulating the necessary commands within a script and enabling automated distribution and execution across worker nodes, we have succeeded in reducing the manual intervention required for node joining. This approach not only optimizes the deployment workflow but also contributes to making the overall process more seamless and less error-prone. While this solution doesn't cover the entire deployment lifecycle, it represents a significant step towards achieving a more automated and efficient Kubernetes deployment process.

We have used the following command to broadcast the script:

- `python -m http.server portnumber`

Multiple clients can access and run the file by running the following command:

- `wget http://masterip:portnumber/filename`
- `./filename`



### 3.3 Automated vertical scaling

In our project, we encountered the challenge that kubeadm lacks support for automatic vertical scaling, a crucial aspect of dynamically adjusting resources in a Kubernetes cluster. To address this limitation, we devised a strategy focused on optimizing resource allocation based on demand. The core concept involves maintaining a minimal number of untainted nodes in the cluster while tainting the remaining nodes. Tainted nodes are untainted only when additional resources are required, ensuring efficient resource utilization.

To implement this solution, we leveraged monitoring tools such as Prometheus and Grafana. Prometheus was employed to collect detailed metrics on resource usage within the cluster, providing valuable insights into its health. We then connected Grafana to Prometheus, which enabled the creation of customized dashboards and the formulation of alert rules.

We established a designated contact point in Grafana, configuring it to utilize a webhook for alert notifications. This webhook is linked to a specific website, and whenever an alert is triggered, Grafana seamlessly initiates a POST request to the configured webhook endpoint. We then created the alert rules in Grafana using multiple queries to trigger notifications to predefined contact points when specific resource thresholds were reached. We wrote a python script to monitor these contact points actively. Upon receiving alerts, the script will dynamically untaint the previously tainted nodes, effectively increasing the available resources in response to heightened demand.

This automated approach streamlines the scaling process, allowing the cluster to adapt to changing workloads without manual intervention. By integrating Prometheus, Grafana, and custom Python scripts, our solution not only monitors the cluster effectively but also orchestrates resource scaling seamlessly.

### 3.4 Distributed storage

In our edge cluster, we implemented distributed storage by installing and deploying Rook Ceph. Rook, a cloud-native storage orchestrator, integrates with Ceph, a robust and scalable distributed storage system, to deliver a reliable and efficient storage solution for our Kubernetes environment. Ceph employs a distributed and redundant architecture, distributing data across multiple nodes to enhance fault tolerance. This ensures high

availability and reliability, which are critical factors for edge environments where nodes may be prone to intermittent connectivity or hardware failures.

After successfully establishing the Ceph cluster in our edge environment, we created a Ceph pool to organize and allocate storage resources efficiently. The pool serves as a logical grouping within the Ceph storage system, defining a set of rules for data distribution and replication. Then, we proceeded to create Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) using a designated storage class and associated them with the configured Ceph pool. This strategic linkage ensures that all PVs are dynamically provisioned and managed by Ceph, establishing a seamless integration between Kubernetes and the Ceph storage infrastructure. By specifying the storage class and associating it with the Ceph pool, we establish an automated process for dynamically provisioning storage resources which not only enhances the agility of our storage infrastructure but also ensures that the underlying Ceph cluster efficiently manages and distributes the storage demands of containerized applications.

### 3.5 Low latency and event driven cluster

Low latency is very important in a Kubernetes cluster for several reasons, especially in scenarios where real-time or near-real-time processing is critical. The main advantages of low latency include the following:

- **Application Responsiveness:** Low latency ensures that applications hosted within the Kubernetes cluster respond quickly to user requests. This is crucial for delivering a seamless user experience, especially in customer-facing applications where delays can lead to dissatisfaction.
- **Edge and IoT Deployments:** In edge and IoT deployments, where Kubernetes clusters may be distributed across geographically dispersed locations, low latency becomes crucial. It ensures that devices at the edge receive timely responses, enabling quick decision-making at the edge without relying on central data centers.
- **Real-time Data Processing:** Applications that require real-time data processing, such as streaming analytics, monitoring, or IoT data ingestion, heavily rely on low latency.
- **High-Performance Computing (HPC):** In scenarios where Kubernetes is used for high-performance computing, such as scientific simulations or data-intensive

calculations, low latency is critical for achieving optimal throughput and reducing computational overhead.

- **Real-time Monitoring and Logging:** Monitoring and logging solutions in a Kubernetes cluster benefit from low latency. Real-time insights into the health and performance of the cluster, applications, and infrastructure components require low-latency data retrieval and analysis.
- **Enhanced User Satisfaction:** Ultimately, low latency contributes to enhanced user satisfaction. Whether users are interacting with web applications, APIs, or other services hosted on Kubernetes, a low-latency environment ensures that responses are swift and in line with user expectations.

### 3.5.1 Implementation:

To achieve a low-latency edge cluster, we have implemented an innovative event-driven and wireless strategy. Prior to deploying any application, we initiate a process where all nodes within the cluster are tainted by the master node. The master node then systematically queries each worker node to gather latency metrics. Leveraging this information, the master node strategically untaints the worker node with the minimal latency, ensuring that subsequent applications are deployed on nodes that are geographically closer and can deliver the lowest latency to end-users. This approach guarantees that users consistently experience minimal latency, contributing to an optimal and responsive application environment.

To address potential fluctuations in resource utilization and maintain low latency, we have integrated Grafana alerts into our system. These alerts are triggered when a node's resource utilization surpasses a predefined threshold. In response, a Python script is automatically executed. This script performs dynamic untainting by identifying the next closest available node and untainting it. Simultaneously, a new instance of the application is seamlessly deployed on this newly untainted node. By automating this process, we ensure that resource-intensive nodes are relieved of excess load, preventing latency spikes and maintaining an efficient and responsive edge cluster. This event-driven approach not only prioritizes low latency but also optimizes resource distribution within the cluster, aligning with the dynamic demands of edge computing.

In summary, our low-latency edge cluster implementation combines strategic tainting for initial node selection based on latency metrics with dynamic untainting triggered by

Grafana alerts. This approach guarantees an event-driven and wireless edge cluster where applications are intelligently placed on nodes with minimal latency. Through continuous monitoring and automation, our system adapts to changing resource conditions, providing users with an optimized and consistently low-latency experience in diverse edge computing scenarios.

## 3.6 Application

We have developed a health care monitoring application within our edge cluster, utilizing Node.js, Express.js, and MongoDB technologies. In this application, we are assuming that a sensor generates data encompassing fields such as name, date, time, and heart rate. This data is then transmitted to a MongoDB pod deployed within the edge cluster.

To facilitate the monitoring process, we've implemented an Express.js code in a separate pod. The application pod is exposed to users through a NodePort service, providing external accessibility. A client can access their heart rate monitoring results using the url `http://masterip:portnumber/patientname`. Kubernetes services are employed to establish connectivity between the Express.js application pod and the MongoDB pod, enabling seamless access to health-related data stored in the "healthcheck" database.

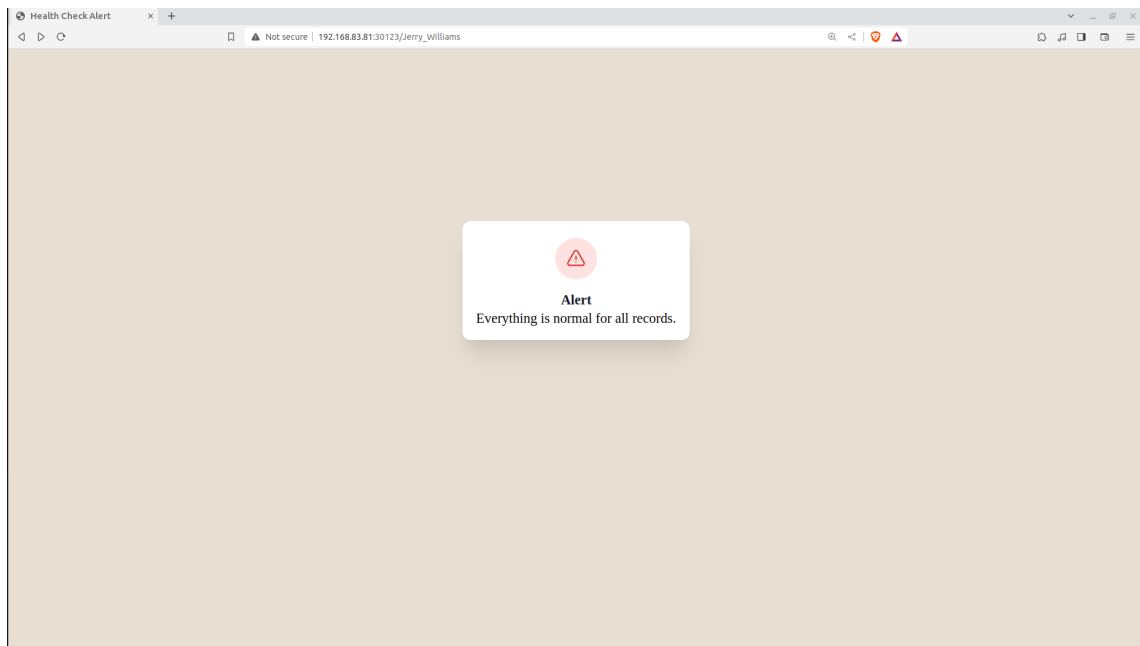


FIGURE 3.2: GUI when heart rate is normal

The monitoring functionality involves assessing the heart rate data of patients. If the latest ten heart rate documents for a patient surpass 100 beats per minute (BPM), indicative of an elevated heart rate, an alert message is dynamically displayed on the graphical user interface (GUI) accessible to clients. Conversely, if the latest ten heart rate documents fall within the normal range (100 BPM or below), a reassuring message indicating that everything is normal is presented on the GUI. Figure 3.2 is the GUI which will be displayed to the users when the heart rate is normal and Figure 3.3 is the GUI which will be displayed when the heart rate is abnormal.

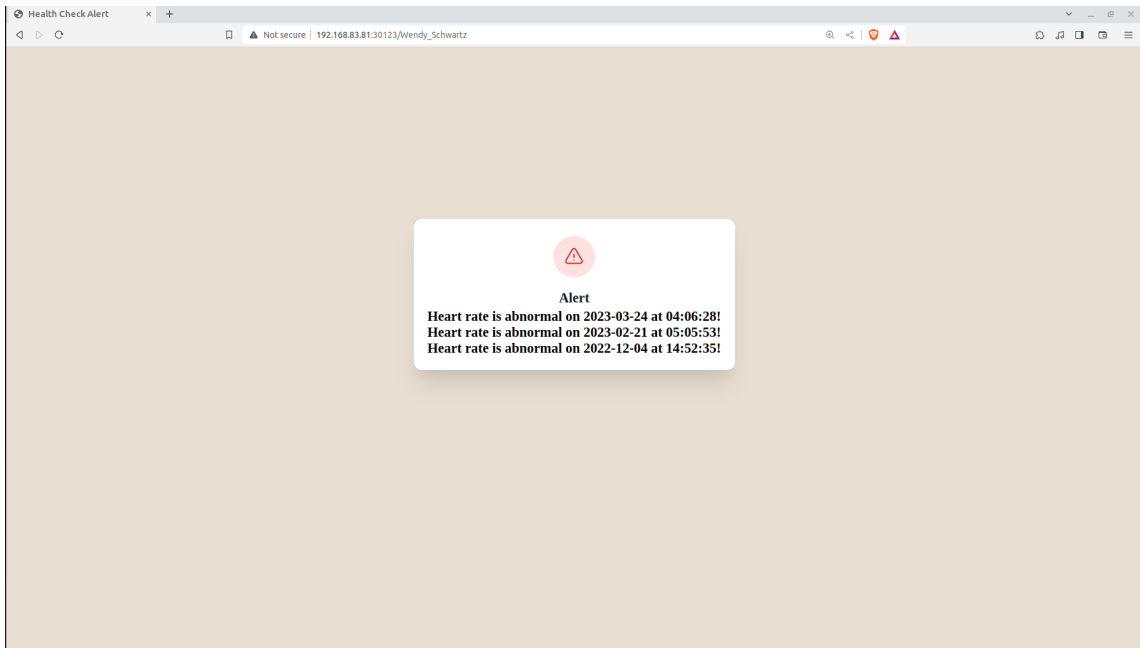


FIGURE 3.3: GUI when heart rate is abnormal

## Chapter 4

# Testing and Results

### 4.1 Use Cases that can benefit by using our edge cluster

Due to the inherent advantages of low latency, event-driven architecture, and wireless connectivity within our edge cluster, the potential applications are diverse and impactful. One notable application is in the realm of drone technology, where the combination of low latency and wireless communication facilitates real-time data processing and decision-making. Drones can leverage the capabilities of the edge cluster for efficient navigation, data analysis, and mission execution, contributing to enhanced performance.

Additionally, the event-driven nature of the edge cluster proves beneficial for ad hoc networks. The dynamic and responsive characteristics of the cluster enable seamless communication and collaboration among devices, making it well-suited for scenarios where network topology may change rapidly or where devices need to connect and disconnect dynamically.

Furthermore, the low-latency environment of the edge cluster is instrumental in supporting critical applications that demand rapid data processing and decision-making. Industries such as healthcare, emergency response, and industrial automation can leverage the edge cluster to ensure timely and reliable execution of critical tasks, enhancing overall system responsiveness.

In summary, the unique attributes of our edge cluster, including low latency, event-driven architecture, and wireless connectivity, make it a versatile and valuable resource for a range of applications. From drone technology to ad hoc networks and critical applications,

the edge cluster stands to significantly enhance the efficiency and capabilities of various systems that benefit from its dynamic and responsive nature.

## 4.2 Testing and Results

### 4.2.1 Testing the application

We tested the cluster ability to manage a substantial influx of requests concurrently. The successful handling of more than 200 simultaneous requests is indicative of the cluster's scalability and responsiveness.

### 4.2.2 Results of multiple features of our cluster

```
root@ramyashri:/home/ramya/app# kubectl get nodes
NAME                                STATUS    ROLES    AGE    VERSION
cantcode-virtualbox                Ready    <none>    32d    v1.28.1
ramyashri                          Ready    control-plane  32d    v1.28.0
sujitha-virtual-machine            Ready    <none>    32d    v1.28.0
ubuntulinux                        Ready    <none>    32d    v1.28.0
root@ramyashri:/home/ramya/app#
```

FIGURE 4.1: Kubernetes Setup

```
root@ramyashri:/home/ramya/app# kubectl get pods
NAME                                READY    STATUS    RESTARTS    AGE
grafana-58d6f67f9-gwbss             1/1     Running    0            33m
grafana-58d6f67f9-hjlg9             0/1     Completed  0            166m
prometheus-alertmanager-0           0/1     Completed  0            3h36m
prometheus-kube-state-metrics-7645bc5dc7-5sq5m  1/1     Running    0            166m
prometheus-prometheus-node-exporter-8bh8r  1/1     Running    0            33m
prometheus-prometheus-node-exporter-9c472  1/1     Running    131 (151m ago)  27d
prometheus-prometheus-node-exporter-q6ddh  0/1     Evicted    0            54s
prometheus-prometheus-node-exporter-qk7m6  1/1     Running    479           27d
prometheus-prometheus-pushgateway-df9c66b44-h2vvt  1/1     Running    0            166m
prometheus-server-f8cbfc66d-tflqm      2/2     Running    0            166m
video-stream-server-57f78c5b58-f9424    1/1     Running    0            166m
root@ramyashri:/home/ramya/app#
```

FIGURE 4.2: Prometheus and grafana pods running in the cluster

```

root@ramyashri:/home/ramya/app# kubectl get pods -n example-applications
NAME                                READY   STATUS    RESTARTS   AGE
health-monitoring-5b44d84c5b-9twcz  1/1     Running   3 (22m ago) 26m
knote-6d887fb64c-mbnm8              1/1     Running   0           168m
mongo-5466974cf-4rlzq               0/1     Completed 0           3h47m
mongo-5466974cf-svs6b               1/1     Running   0           35m
mongo-heart-6c8f9b9954-q487l        1/1     Running   0           3h45m
root@ramyashri:/home/ramya/app#

```

FIGURE 4.3: Applications we have tried on our cluster

```

root@ramyashri:/home/ramya/app# kubectl get pods -n rook-ceph
NAME                                READY   STATUS    RESTARTS   AGE
csi-cephfsplugin-2t8vw              2/2     Running   12          9d
csi-cephfsplugin-48w7t              2/2     Running   4           9d
csi-cephfsplugin-provisioner-55588874-gwdpw  5/5     Running   7 (154m ago) 3h49m
csi-cephfsplugin-provisioner-55588874-vznsr  5/5     Running   0           170m
csi-cephfsplugin-x4fsf              2/2     Running   6           9d
csi-rbdplugin-br6k9                 2/2     Running   12          9d
csi-rbdplugin-provisioner-577dff4756-bcvc5  5/5     Running   0           170m
csi-rbdplugin-provisioner-577dff4756-tfvjn  5/5     Running   7 (154m ago) 3h49m
csi-rbdplugin-v4jkg                 2/2     Running   4           9d
csi-rbdplugin-zhtws                 2/2     Running   6           9d
rook-ceph-crashcollector-cantcode-virtualbox-677d8c9dc9-mnqtq  1/1     Running   0           36m
rook-ceph-crashcollector-sujitha-virtual-machine-79978ff99kvtms  1/1     Running   0           3h54m
rook-ceph-crashcollector-ubuntulinux-6795c5df6b-7bvd1  1/1     Running   0           36m
rook-ceph-mds-myfs-a-bd79884f4-wqptr      2/2     Running   0           157m
rook-ceph-mds-myfs-b-886c9884-2hfp7       2/2     Running   8 (152m ago) 3h51m
rook-ceph-mgr-a-f6ff849c8-4mkhd           3/3     Running   0           157m
rook-ceph-mgr-b-5f44878665-4r9xb          3/3     Running   1 (3h43m ago) 3h51m
rook-ceph-mon-a-cdbc787fb-xbsz5           2/2     Running   0           3h49m
rook-ceph-mon-b-7d69894f59-pfxvr          2/2     Running   0           3h46m
rook-ceph-mon-c-557bdbbd5c-5t86c          2/2     Running   0           170m
rook-ceph-operator-6c56647556-wc4bl       1/1     Running   0           170m
rook-ceph-osd-prepare-cantcode-virtualbox-twjh6  0/1     Completed 0           34m
rook-ceph-osd-prepare-sujitha-virtual-machine-s5s94  0/1     Completed 0           34m
rook-ceph-osd-prepare-ubuntulinux-rs7n4     0/1     Completed 0           34m
rook-ceph-tools-84f9854d5f-rlj8b          1/1     Running   0           157m
root@ramyashri:/home/ramya/app#

```

FIGURE 4.4: Pods that are running in rook ceph

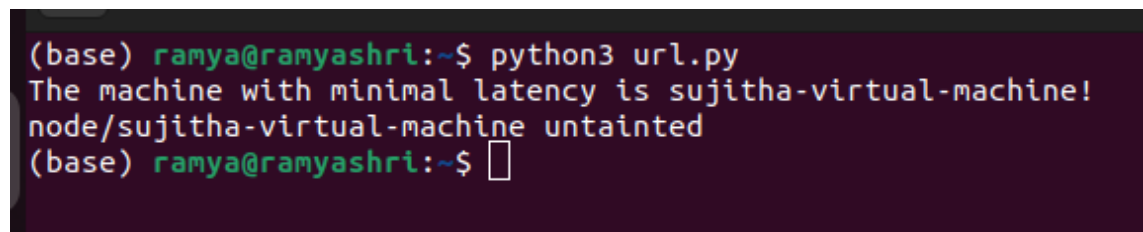
```

root@ramyashri:/home/ramya/app# kubectl get pods -n kube-system
NAME                                READY   STATUS    RESTARTS   AGE
coredns-5dd5756b68-45zcb            1/1     Running   1           36h
coredns-5dd5756b68-g5xs9            1/1     Running   1           36h
etcd-ramyashri                       1/1     Running   1633 (47m ago) 32d
kube-apiserver-ramyashri              1/1     Running   2588 (44m ago) 32d
kube-controller-manager-ramyashri     1/1     Running   33          32d
kube-proxy-7sv7p                     1/1     Running   3           32d
kube-proxy-h5vct                     1/1     Running   17          32d
kube-proxy-r59n8                     1/1     Running   8           32d
kube-proxy-r5fcd                     1/1     Running   12          32d
kube-scheduler-ramyashri              1/1     Running   35          32d
root@ramyashri:/home/ramya/app#

```

FIGURE 4.5: Pods that are deployed once the kubernetes is created.





```
(base) ramya@ramyashri:~$ python3 url.py
The machine with minimal latency is sujitha-virtual-machine!
node/sujitha-virtual-machine untainted
(base) ramya@ramyashri:~$
```

FIGURE 4.6: Untainting low latency node to deploy the application

## Chapter 5

# Summary and Future Directions

### 5.1 Conclusion

This project represents the successful implementation of a cutting-edge, wireless, event-driven, fault-tolerant, and low-latency edge cluster. Leveraging a combination of advanced technologies such as Kubernetes, Docker, Grafana, Prometheus, Node.js, MongoDB, and Ceph, we've developed a versatile platform that can be harnessed across various domains and applications. The utilization of Kubernetes ensures efficient orchestration, while Docker facilitates containerization for seamless deployment and management. Grafana and Prometheus provide robust monitoring capabilities, enhancing fault tolerance, and Node.js with MongoDB contribute to the development of responsive, event-driven applications.

While the low-latency aspect has been implemented partially, the foundation has been laid for future optimizations to achieve even greater efficiency and responsiveness. The edge cluster, as it stands, already offers significant advancements, providing novel solutions to challenges commonly associated with edge computing, especially in Kubernetes (K8S) environments. The automatic and efficient deployment mechanisms, coupled with automatic vertical scaling capabilities, showcase the cluster's adaptability to fluctuating workloads.

## 5.2 Future Works

While our current edge cluster has implemented the low-latency feature to some extent, our aim is to elevate this capability to a higher standard by fundamentally redefining the architecture. In our future plans, we aim to reconfigure the edge cluster architecture itself by selecting nodes with minimal latency. This strategic shift involves excluding nodes with higher latencies, thereby optimizing the overall cluster performance.

In addition to it, we intend to employ more optimal algorithms for identifying nodes with minimal latency, moving beyond our existing methodology. This shift towards algorithmic precision ensures a more accurate and efficient selection process, aligning with our objective of achieving the lowest possible latency across the cluster.

Furthermore, recognizing the importance of scalability, we plan to augment the resources of the cluster substantially. This expansion will enable us to conduct comprehensive tests on a larger and more complex scale, providing insights into the cluster's performance under diverse conditions.

In addition to refining our low-latency implementation, we would like to explore the concept of mobility within the context of wireless nodes. This exploration recognizes the dynamic nature of edge computing environments, where nodes may move, connecting and disconnecting seamlessly. Our focus extends beyond conventional static deployments, as we aim to understand and optimize the latency implications associated with the mobility factor.

In essence, our future endeavors involve not only refining the selection of low-latency nodes but also expanding the cluster's capacity to accommodate more extensive testing scenarios. Through these strategic initiatives, we aim to establish a highly efficient and responsive edge cluster that excels in delivering low-latency and fault tolerant solutions across a variety of applications and use cases.

## Chapter 6

# References

- Docker Official Documentation - <https://docs.docker.com/get-started/>
- Kubernetes official Documentation - <https://kubernetes.io/docs/home/>
- Grafana - <https://grafana.com/docs/>
- Prometheus - <https://prometheus.io/docs/prometheus/latest/docs>
- Ceph rook - <https://rook.io/docs/rook/latest-release/Getting-Started/intro/>
- Nodejs - <https://nodejs.org/api/>
- MongoDB - <https://www.mongodb.com/docs/>
- For pictures:
  - <https://geekflare.com/kubernetes-architecture/>