

EVALUATING KUBERNETES AT THE EDGE FOR FAULT TOLERANT
MULTI CAMERA COMPUTER VISION APPLICATIONS

by

Owen Heckmann

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Computer Engineering

Charlotte

2022

Approved by:

Dr. Arun Ravindran

Dr. Hamed Tabkhi

Dr. James Conrad

ABSTRACT

OWEN HECKMANN. Evaluating Kubernetes at the Edge for Fault Tolerant Multi Camera Computer Vision Applications. (Under the direction of DR. ARUN RAVINDRAN)

The rise of AI powered computer vision algorithms offers the possibility of intelligent data processing, and decision making based on streaming data from video cameras. Applications such as Smart Cities could potentially use these video cameras for a variety of use cases such as pedestrian detection, public safety, and traffic monitoring. The requirements for low latency for real-time decision making, and the privacy needs of video data, leads to use of edge computing to process raw video frames. However, unlike cloud computing with almost unbounded resources, the edge is characterized by compute nodes of limited capacity and power budget. Additionally, fault tolerance is limited due to replication costs at the edge.

In this thesis, we investigate the design of a fault tolerant edge cluster consisting of low power ARM based Raspberry Pi 4 nodes. In the cloud, Kubernetes is used as a system orchestrator for large clusters. An edge tailored version of Kubernetes, K3s has recently been made available. However, prior research has not characterized the resource consumption and latency impact of K3s on realistic edge clusters. In this thesis we fill the gap by an extensive evaluation of K3s at the edge on our Raspberry Pi 5 cluster. Our results indicates that while K3s does add significant resource and latency overhead to edge applications, it still delivers on fault tolerance at the edge.

DEDICATION

To my family, who never stopped believing that I could do it.

ACKNOWLEDGEMENTS

I would like to acknowledge and thank Dr. Arun Ravindran for advising me, and working with me at every step in my Masters degree. I couldn't have done it without you.

I would also like to thank my committee members, Dr Conrad and Dr. Tabhki for joining my committee.

Finally, I would like to thank Babak Ardabili and the rest of his team for offering me a job at the smart cities project, which started me on the problem which became this thesis.

TABLE OF CONTENTS

LIST OF FIGURES	viii
CHAPTER 1: Introduction	1
1.1. Introduction	1
CHAPTER 2: Background	5
2.1. Background	5
2.2. Edge Computing	5
2.3. Computer Vision	6
2.4. Edge gateway	7
2.5. Fault Tolerance at the Edge	8
2.6. Kubernetes	9
2.7. K3s	12
2.8. Prometheus and Grafana	13
2.9. Related Work	14
CHAPTER 3: Experimental Design	15
3.1. Design	15
3.2. Hardware	15
3.3. Installing K3s	17
3.4. NATS in K3s	18
3.5. Containerizing Application	18
3.6. Running Containers in K3s	19
3.7. Monitoring overhead of system in K3s	20

3.8. Measuring baseline system	21
CHAPTER 4: Experimental Results	22
4.1. CPU and memory utilization	22
4.2. Latency	25
4.3. Characterization of Node Failure	28
CHAPTER 5: Discussion	34
CHAPTER 6: Conclusion	36
REFERENCES	37
APPENDIX A: Configuration Scripts for K3s	39
A.1. GRPC healthcheck Dockerfile	39
A.2. GRPC Healthchecks in K3s	39
A.3. K3s YAML	40
A.4. VEI Service YAML	41
A.5. Emulated Vision Application YAML	42
A.6. Reboot Bash Script	43
A.7. Monitoring system YAML	44
A.8. Prometheus YAML	46
A.9. Instrumentation for VEI in Golang	47
A.10Instrumentation for Emulated Vision Application in Python	48
A.11Configuration File for Standalone Prometheus	49

LIST OF FIGURES

FIGURE 1.1: Edge-Cloud Platform for Pedestrian Monitoring. IoT camera nodes equipped with embedded GPUs perform deep learning based vision Processing, with output published to edge server. Analytics application at edge server subscribes to vision processing output, and publishes results to the cloud. From the cloud, relevant event sent to end users via a mobile application.	2
FIGURE 2.1: Cloud paradigm for IoT	6
FIGURE 2.2: Edge computing paradigm for IoT	7
FIGURE 2.3: Block diagram of the VEI edge gateway	8
FIGURE 2.4: Structure of Kubernetes. the services which control the cluster as a whole run on the Master Nodes: etcd, which stores information about the desired state of the cluster, the Scheduler, which tells Worker Nodes to deploy services, and the Controller Manager, which gives instructions to Worker Nodes about actions. Meanwhile each worker node contains: Kubelet, which executes instructions given to it by the Master Node, the Kube-proxy, which mediates communications between containers and end users, and the objects, running on the system as a whole. Communication between the components and the developer are handled by the API-server, running on the Master Node Source: [21]	11
FIGURE 2.5: Structure of Kubernetes in High Availability Mode. There are multiple master nodes, with redundant databases and control services, as well as multiple worker nodes. Communication between them is mediated via a load balancer. Source: [17]	12
FIGURE 2.6: Structure of k3s Nodes. k3s Server stores data about the desired state in the edge cluster with SQLite, schedules when and where to deploy pods with the scheduler, and controls the state of the Agent Nodes with the Controller Manager. The k3s agent [26]	13
FIGURE 3.1: Baseline edge system	16
FIGURE 3.2: Picocluster edge testbed	16
FIGURE 3.3: Screenshot of Pi4 nodes running in K3s cluster	17

FIGURE 3.4: Screenshot of NATS running on K3s in High Availability mode with 3 Replicas	18
FIGURE 3.5: All VEI Components running in K3s in High Availability mode with 3	19
FIGURE 4.1: Baseline System Resource Consumption	23
FIGURE 4.2: Resource consumption of system in K3s	25
FIGURE 4.3: Cumulative Distribution Function (CDF) of latency (milliseconds) for baseline edge system. The latency CDF was plotted over 1000 frames at a rate of 10 FPS	26
FIGURE 4.4: Cumulative Distribution Function (CDF) of VEI latency (milliseconds) for edge system in K3s. The latency CDF was plotted over 1000 frames at a rate of 10 FPS	27
FIGURE 4.5: Failure of Node with only NATS	28
FIGURE 4.6: Failure of node with emulated vision application	29
FIGURE 4.7: Failure of node with VEI edge gateway	30
FIGURE 4.8: Failure of node with VEI and emulated vision application	30
FIGURE 4.9: Failure of node with NATS and the emulated vision application	31
FIGURE 4.10: Failure of node with NATS and VEI	32
FIGURE 4.11: Characterization of failure of node with NATS, VEI, and the emulated vision application running	32

CHAPTER 1: Introduction

1.1 Introduction

The term Smart Cities is used to describe the application of smart devices to the efficient running of a city. Smart Cities technologies include a variety of city-wide sensors, Internet-of-Things (IoT), and Artificial Intelligence to collect, analyze, and apply information about a city in real time. With the application of Big Data analysis and AI, stakeholders can make better informed decisions, leading to several positive outcomes such as better quality of life, improved environmental outcomes, operational efficiency and lower city operational costs [5].

Among the sensors available for data collection, visual data captured by video cameras represent a rich source of information. Many cities worldwide have video cameras already installed for tasks such as traffic monitoring. These can be repurposed for other uses such as public safety and environmental monitoring. Traditionally, much of this monitoring was done manually by reviewing the video recording after the event of interest had occurred. Recent advances in deep learning based vision algorithms, and the availability of highly scalable computational resources in cloud computing platforms have made it possible to automatically analyze events of interest in a video stream [19]. However, large data sizes of video streams imposes constraints on transmitting via the internet to a cloud provider [23]. Additionally, the added latency may not be appropriate for real-time processing. Furthermore, privacy constraints and data privacy laws limit the transmission of information rich video data to a distant cloud.

The edge computing paradigm involves performing the computation close to the source of the data. This reduces the network bandwidth requirements, lowers latency,

and keeps the data in the privacy perimeter of the end user (for example, home, office, municipal jurisdiction etc.) [20]. The availability of embedded accelerators has made deep learning based vision algorithms practical at the edge [23]. Only the results of the computation (for example, detected objects) rather than raw video frames needs to be transmitted to the cloud, for further analysis. Figure 1.1 shows such an edge-cloud platform for pedestrian monitoring.

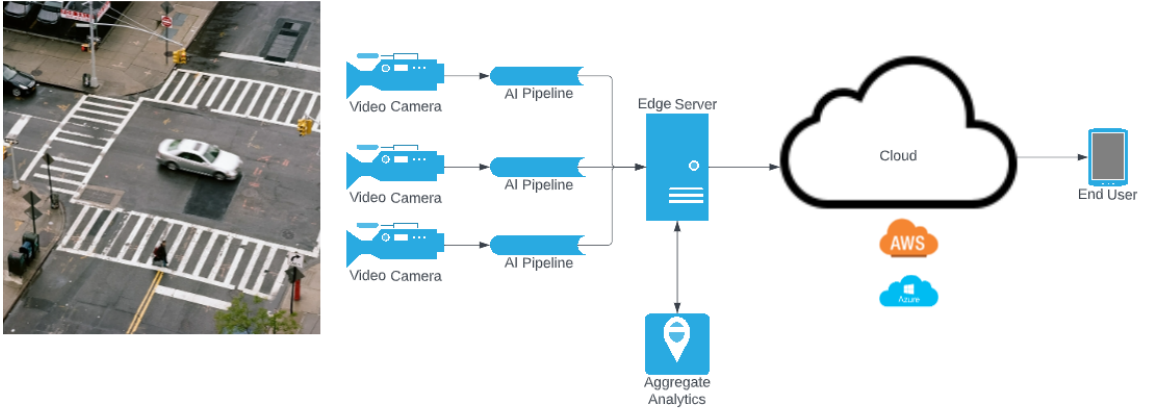


Figure 1.1: Edge-Cloud Platform for Pedestrian Monitoring. IoT camera nodes equipped with embedded GPUs perform deep learning based vision Processing, with output published to edge server. Analytics application at edge server subscribes to vision processing output, and publishes results to the cloud. From the cloud, relevant event sent to end users via a mobile application.

Despite the promises offered by edge computing for computer vision, several technical challenges need to be tackled to make the edge computing close to providing the same capabilities as the cloud. Some of these include fault tolerant computing, resource scaling, and resource allocation models such as Infrastructure-As-Code, and serverless computing [20].

In this thesis, we focus on the problem of fault tolerance at the edge. In the cloud, fault tolerance is achieved through a cluster of servers with computing services replicated across them, such that the system remains operational even if one or more servers fail. Kubernetes, originally open-sourced by Google, is used in orchestrating

this cluster [13]. Tasks such as scheduling jobs on nodes in the cluster, monitoring the nodes for failures, moving computing tasks away from failed nodes, and automatically managed by Kubernetes [21]. This frees the application programmers from fault tolerance issues, as long as their applications are written to run on Kubernetes. However, Kubernetes is a complex resource intensive systems, that often requires a team of dedicated engineers to keep operational. Recognizing that the stock Kubernetes may not be suitable for the edge, the company Rancher (since acquired by SUSE), has made available a stripped down edge friendly version of Kubernetes called K3s [26]. However, a thorough investigation of K3s on a realistic edge platform in terms of its impact on application performance, and system resource usage is missing in the the literature.

In this thesis, we undertake an experimental evaluation of K3s on a Raspberry Pi 4 (Pi4) based edge Pico cluster. The cluster has 5 Pi 4 nodes with a total power consumption of 30 W making it an attractive edge computing platform. The application that runs on this cluster is an edge gateway that supports multiple vision applications operating on one or more video streams, with multiple cloud backends.

The thesis makes the following contributions -

- Characterizes K3s resource consumption on Pi4 based edge cluster
- Evaluates added application latency due to K3s
- Evaluates the latency impact of recovery on node failure under multiple failure scenarios.

The thesis is organized as follows. Chapter 2 provides a brief background on edge computing, computer vision, and Kubernetes including K3s. Chapter 3 describes the setup of the experimental platform including the necessary steps needed to run an edge application on Kubernetes. Chapter 4 presents the experimental evaluation of K3s on the Pi4 based edge cluster. Chapter 5 discusses the suitability of K3s for edge

computing based on our experimental results. Chapter 6 concludes the thesis with a summary of results, and directions for future work.

CHAPTER 2: Background

2.1 Background

In this chapter, we provide a brief background on edge computing, computer vision, Kubernetes, K3s, fault tolerance at the Edge, and related work on this Topic.

2.2 Edge Computing

Edge computing emerged as a paradigm with the increasing size and complexity of IoT networks [20]. Under the cloud computing paradigm, as the size of IoT networks increase, the limitations increasingly become apparent [27]. First, as more data is produced at the edge, sending data to the cloud requires more bandwidth. However, the increase in bandwidth available to IoT networks has not grown as fast as the amount of data produced [1] [6]. Second, the sending raw, or unprocessed data to the cloud adds latency to the system. At a certain level of system latency, it becomes infeasible to use the data in real time. Third, as the IoT systems proliferate, they place an increasing strain on the bandwidth and reliability of current networks. 2.1 shows the structure of the conventional cloud computing paradigm. Data producers - such as sensors - generate data, which is then uploaded to the cloud [4]. Any processing of the data occurs at the cloud. Data consumers request data from the cloud, and the cloud pushes the data to the data consumers [20].

In the edge computing paradigm, applications that are usually run on the cloud are instead brought closer to the data sources on the edge. Here, "Edge" is defined as as the computing and network systems which are between the data sources and the cloud. Edge computing takes advantage of the rapid growth in computer processing power relative to bandwidth to process data locally, and then uploading the results

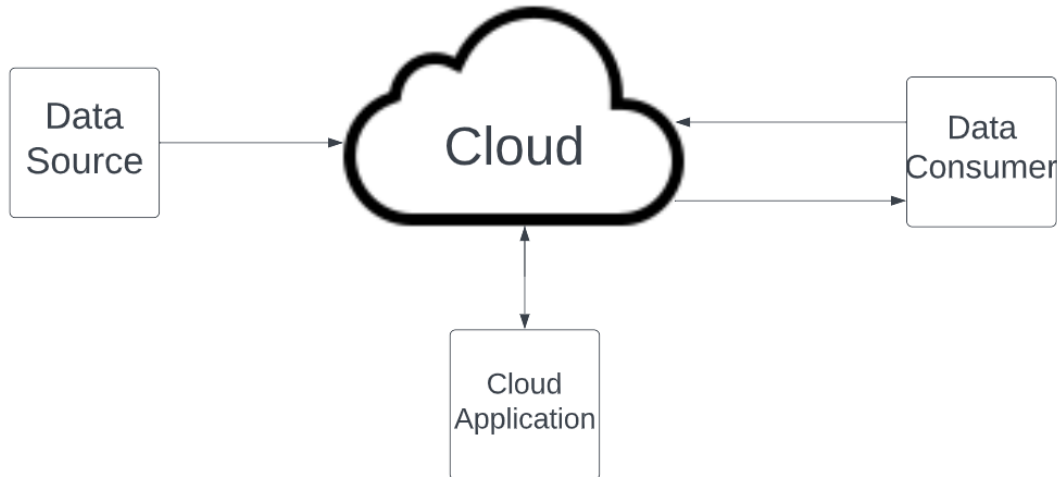


Figure 2.1: Cloud paradigm for IoT

to the cloud [4]. Doing so reduces the bandwidth needed to for the connection to the cloud, which reduces the latency of the upload [20]. Therefore, though processing at the edge may take longer than at the cloud due to potentially less powerful compute processors, the system as a whole has less latency under the edge paradigm. Figure 2.2 shows the structure of the Edge computing paradigm. Data producers - such as sensors - generate data, which is then processed at the edge. The results of the processed data is uploaded to the cloud [27]. The data consumers request data from the cloud, and the cloud pushes the data to the data consumers if they are not local to the edge. In that case, they request data from the edge instead [20].

2.3 Computer Vision

Computer Vision is a field that attempts to create computer systems that can understand and interpret images [12]. Though images can provide much information, creating computer systems that can extract this data traditionally been difficult. This is problematic, as images taken by cameras are often the best way to find information about an environment. Without a robust computer vision solution, extracting data for computer systems to use has required human assistance. Unfortunately, human assistance is expensive, and difficult to effectively scale. With the advent

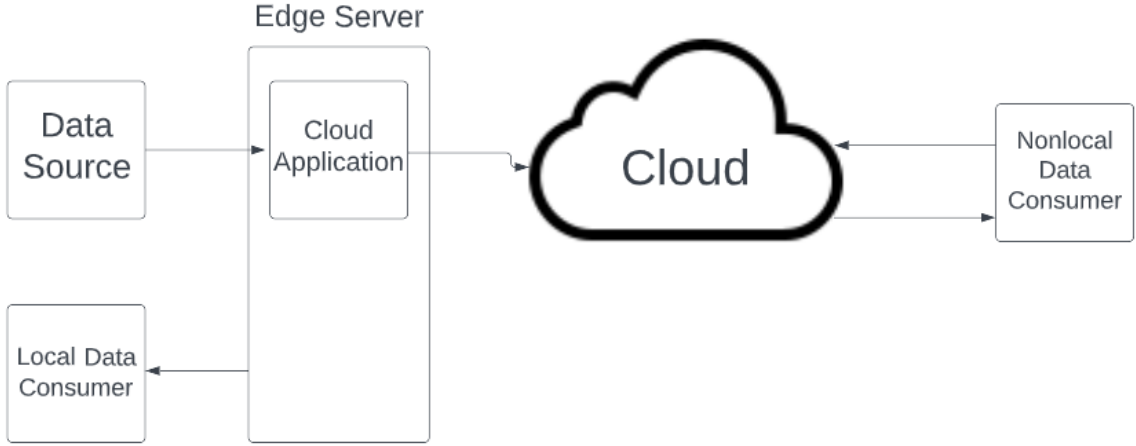


Figure 2.2: Edge computing paradigm for IoT

of deep learning algorithms, the number and effectiveness of computer vision tools and techniques have undergone a period of growth [11]. Many of these computer vision applications have succeeded in reliably extracting and classifying features from images, which was previously a difficult task.

An example of current computer vision algorithms is a series of algorithms called You Only Look Once, or YOLO. YOLO uses Convolutional Neural Networks to extract and classify objects in an image [19]. Over its development from YOLO v1 to YOLO v7, it has underwent significant improvement, both in effectiveness and efficiency. [16]

In the smart cities context, video cameras offer the capability to monitor multiple events in a city. Traditionally, public safety has involved monitoring humans to detect unsafe events. However, with robust computer vision applications, this process could be automated, with concomitant gains in safety and efficiency. Therefore, the ability to run computer vision algorithms efficiently and reliably is an important task.

2.4 Edge gateway

Visual Edge IoT, or VEI is a multicloud edge gateway proposed by Luu et al. [23] targeted specifically at IoT computer vision applications with high bandwidth

requirements and low tolerances for latency. VEI enables decoupling of cameras from vision applications running at the edge. This allows a single application to consume video streams from multiple cameras, as well as multiple applications to consumer video streams from a single camera. [10].

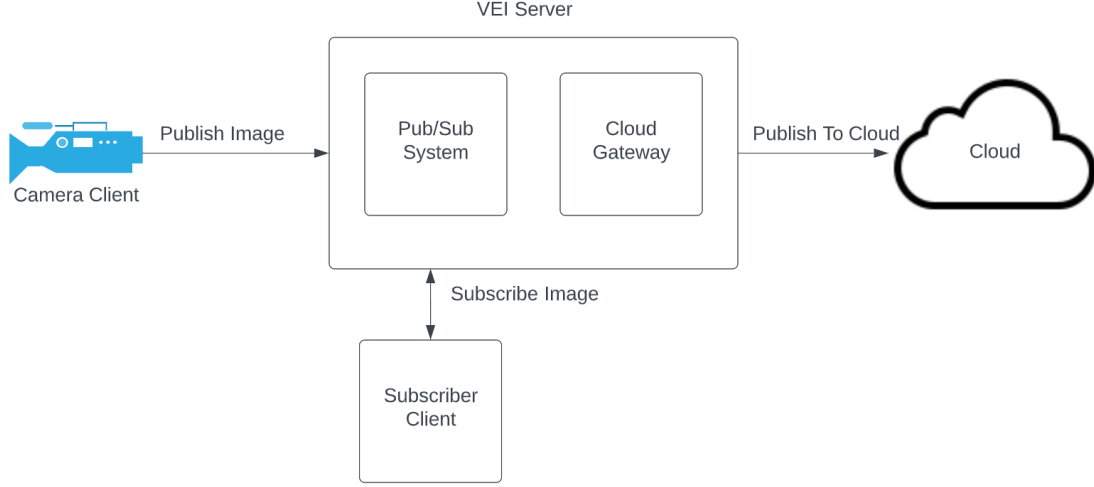


Figure 2.3: Block diagram of the VEI edge gateway

VEI edge gateway consists of four components: A server implemented in Golang, a pub/sub system using an open-source messaging server called NATS, and camera and subscriber client implemented in Python[10] [24]. The APIs needed to communicate between the components are implemented as (Remote Procedure Calls) RPCs using *gRPC*. Figure 2.3 shows the block diagram of VEI, as well as its external clients.

When in operation, the camera client publishes video frames via the VEI API to the VEI server. The subscriber, in turn, receives the images and analyzes them. The subscriber then publishes the results of this analysis to the cloud[10].

2.5 Fault Tolerance at the Edge

In real world applications, it is almost inevitable that faults will occur, in both hardware and software. Therefore, it is desirable that implemented systems can tolerate these faults when they occur. Different components of the IoT edge system have

different fault tolerances. For instance, the cloud component of the system has the highest fault tolerance. Fault tolerance at the cloud is implemented by cloud vendors taking via the large resource availability in cloud data centers. The component of the system with the least fault tolerance is the edge device, and the applications that run at the edge.

Both the VET edge gateway and the subscriber client run on a single piece of hardware. This single piece of hardware forms a single point of failure. Similarly, the system as a whole also fails if there is a software failure such as program crash. Human intervention is needed to restart the edge server.

A desirable solution to the problem of fault tolerance at the Edge should have several characteristics. First, the solution should be able to run on conventional edge hardware. Similarly, the solution should impose a reasonable cost in terms of overhead and latency. Additionally, the system should be able to recover upon failures without human intervention.

A concept for a solution that has these characteristics would be to have multiple replicas of the edge software running on multiple edge hardware nodes. One of the edge nodes, and the software replica would be nominated as the primary. If either the edge node or the software replica fails, then the system could recover from the fault via the backup replicas. The video frames from the cameras would be rerouted to the new primary. The subscriber client would still be able to receive the information that they had subscribed to. All of the data would still be uploaded to the cloud, with little or no interruption of service.

2.6 Kubernetes

Kubernetes, or K8s is a container orchestration system for clusters of distributed hardware. It aims to provide automated ways to manage distributed components and services across varied and distributed hardware. Containers are light-weight OS level virtual machines consisting of programs and their dependencies, running in a

self-contained environment [7]. Kubernetes enables automatic deployment, scaling, and restarting containers across the distributed hardware in cloud data centers [13].

In Kubernetes, different physical or virtual machines are abstracted into nodes. Nodes can be connected together into sets called clusters. In a cluster, some nodes are designated as Master Nodes, which has additional software which controls behavior across the cluster [17]. The other nodes are designated as Worker Nodes, which only have the software which controls the behavior of the containers which are running on it [9]. A cluster needs to have at least one Master node, but can have multiple worker nodes. However, if all of the Master nodes fail, then the cluster will fail as a whole [25]. Typically, the master is replicated across 3 nodes.

To operate Kubernetes, the user specifies a desired state for the cluster using YAML configuration files. The specification defines how many replicas of which programs should be running, along with which applications they use, which resources should be made available for them, and other configuration details[21]. Kubernetes will then automatically alter the state of the Cluster to match with the specification [25]. Kubernetes manages containers as pods of one or more containers. Pods represent the basic scheduling unit for Kubernetes.

For instance, if the user specifies that three copies of a pod should be running, Kubernetes will start 3 pods on different nodes throughout the cluster. If one of those containers fails, Kubernetes will detect that the container has failed, and then restart it. If the user updates the program running in the containers, Kubernetes will restart them one at a time. That way, the system will continue to be available, while being updated. If the user specifies that 5 copies of the should run, Kubernetes will start two more containers. If the users specifies that only 1 copy should run, Kubernetes will gracefully stop all containers but one. Kubernetes will also attempt to deploy the different containers across the cluster in a way that is both efficient and fault tolerant[21].

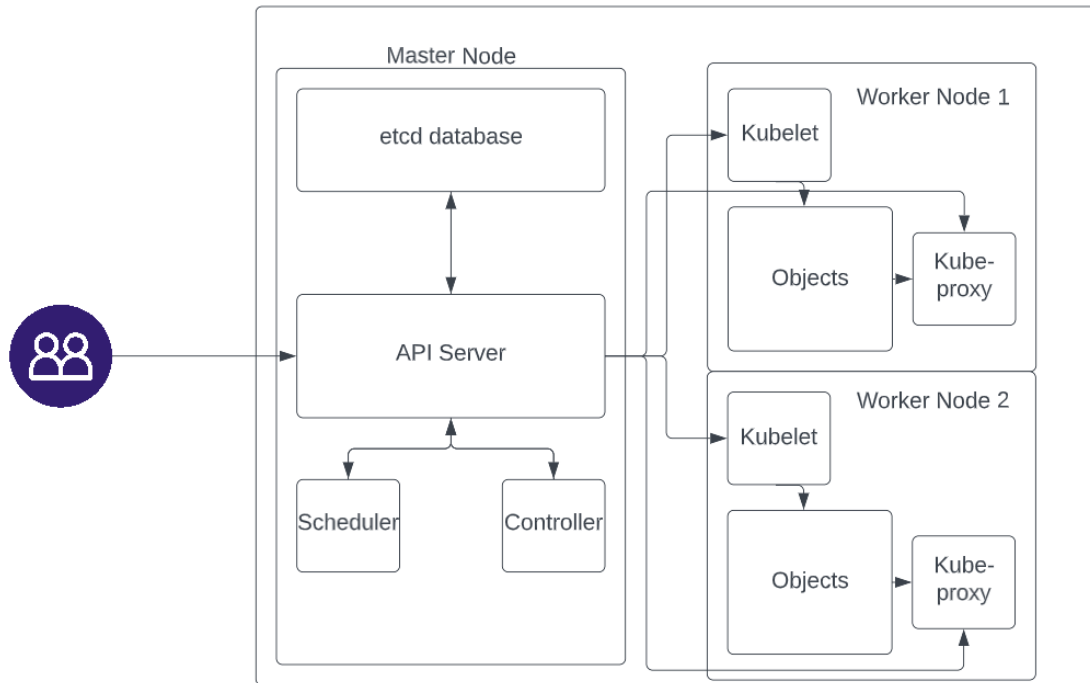


Figure 2.4: Structure of Kubernetes. the services which control the cluster as a whole run on the Master Nodes: etcd, which stores information about the desired state of the cluster, the Scheduler, which tells Worker Nodes to deploy services, and the Controller Manager, which gives instructions to Worker Nodes about actions. Meanwhile each worker node contains: Kubelet, which executes instructions given to it by the Master Node, the Kube-proxy, which mediates communications between containers and end users, and the objects, running on the system as a whole. Communication between the components and the developer are handled by the API-server, running on the Master Node Source: [21]

Kubernetes also abstracts container's communications to services[17]. All of the containers of the same type will be connected to a service. Other programs can connect to the service through its DNS, which automatically handles connections to the containers[17]. If a container fails, then the service will not route connections to it. The user may also specify the behavior of the service. For instance, the service can be specified to balance the incoming communication between containers to avoid overloading any container[21].

However, Kubernetes has costs associated with it as well. An application that is running in a container uses more resources than the same application running on bare

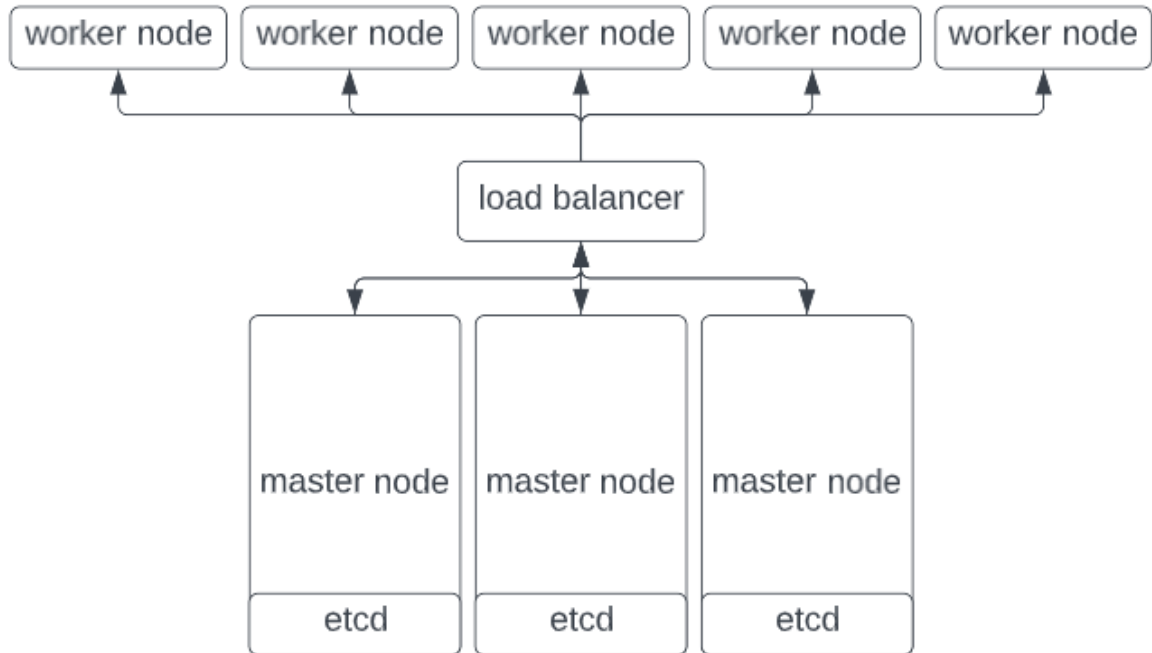


Figure 2.5: Structure of Kubernetes in High Availability Mode. There are multiple master nodes, with redundant databases and control services, as well as multiple worker nodes. Communication between them is mediated via a load balancer. Source: [17]

metal. K8s uses additional resources still needed to run its components. Replicating the containers also means that they use more resources too. Also, because communications in a containerized system entail using a service as an intermediate layer, there is also inherently more latency. All of these cost impose additional difficulties on designing and building fault tolerant systems using Kubernetes at the edge.

2.7 K3s

K3s is a version of Kubernetes created by Rancher specifically designed for the edge. In particular, it is built for ARM architecture in addition to x86 architectures[26]. Because it is built for edge applications, it is significantly less resource hungry than Kubernetes. K3s is installed as a binary that is only 54 Megabytes in size [26]. When deployed on a Master Node, called a Server in k3s, it requires only 512 megabytes of RAM. If deployed on Worker Node, called an Agent, it requires only 256 megabytes

of RAM to run [2].

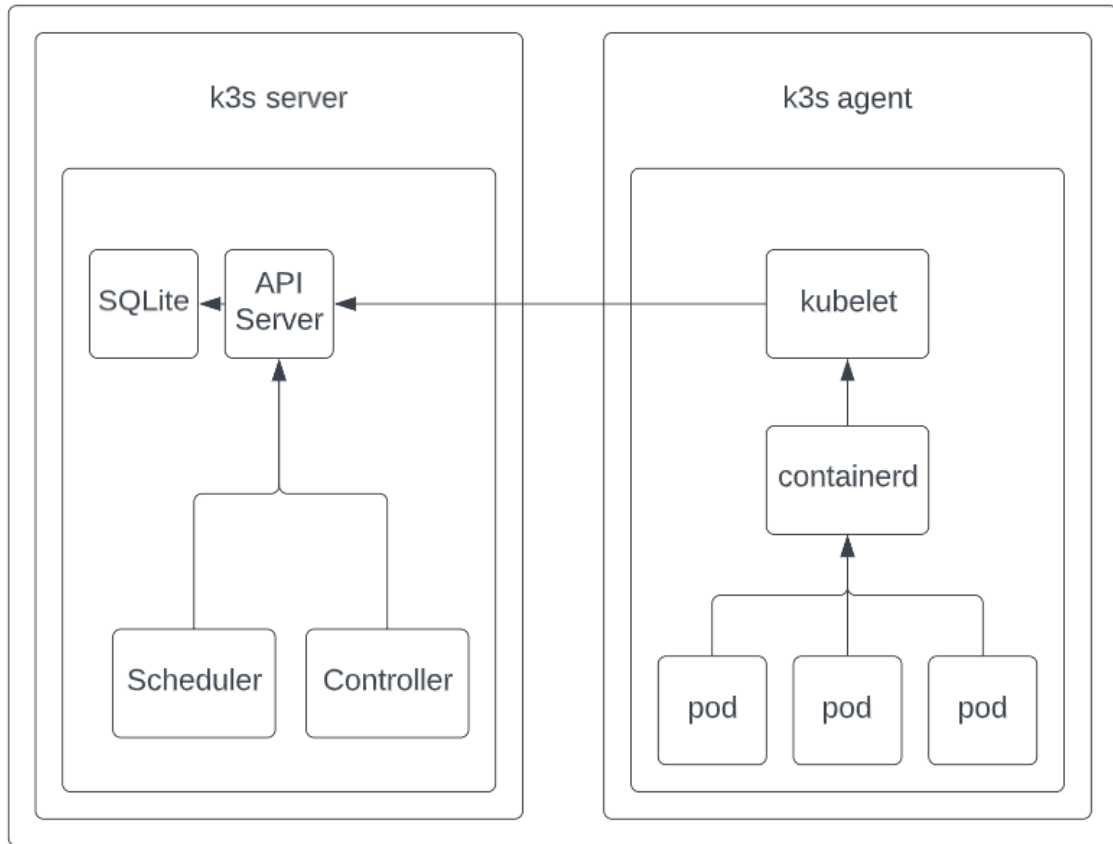


Figure 2.6: Structure of k3s Nodes. k3s Server stores data about the desired state in the edge cluster with SQLite, schedules when and where to deploy pods with the scheduler, and controls the state of the Agent Nodes with the Controller Manager. The k3s agent [26]

2.8 Prometheus and Grafana

Prometheus is an open-source software application that is used for monitoring different events that occur on a system. Prometheus measures numerical information about a system, or metrics. To monitor metrics about some program, the program is modified, or instrumented to provide metrics for a Prometheus application to read. Prometheus then stores these metrics in the form of a time series. These metrics can then be retrieved for viewing over some period of time using a query language called PromQL. Prometheus applications can be run both as a standalone application, or on a K3s Cluster.

Grafana is an open-source web application for data analytics and visualization. Grafana is often used in addition to Prometheus to display the results of multiple PromQL queries in more user-friendly ways.

2.9 Related Work

Previous research have attempted to evaluate the usefulness of container orchestration Systems for IoT Applications. Fathoni et al. compared the difference in RAM and CPU consumption between two different lightweight Kubernetes distributions, specifically KubeEdge and K3s [10]. They found that there was no significant difference between the two in terms of resource consumption. However, they did not attempt to characterize the system while it was running, merely while in an idle state. Eiermann et al also characterized the Memory and CPU consumed by pods in a cluster. They characterized the resources consumed in a five node cluster, but also only during an idle state [8]. Böhm and Wirtz comprehensively analyzed the resource consumption of Kubernetes, MicroK8s, and K3s during different parts of the cluster lifecycle [3]. They found that K3s had on average the best performance across the different parts of the cluster lifecycle. However, they generated their benchmarks using a cluster of virtual machines running on a single physical machine. Finally, Leskinen analyzed the usefulness of K3s in an industrial IoT context [22]. However, none of these offer thorough investigation of K3s on a realistic edge platform in terms of its impact on application performance, and response to node failure.

CHAPTER 3: Experimental Design

In this section, we present the design and implementation of our edge cluster, VEI edge gateway, subscriber clients, and monitoring software.

3.1 Design

We tested the deployment of an example edge application to K3s to characterize its real overhead and fault tolerance. The system architecture of our system is as is shown in 3.1. In the baseline system, a camera streams images to our edge server on running VEI edge gateway, and an emulated vision application all running on a single Raspberry Pi 4 (Pi4) node. The emulated vision applications subscribes to video frames from VEI and publishes fictitious data to the cloud. The emulation exercises the APIs without consuming significant resources such as real computer vision application such as YOLO would have done. The fault tolerant version has all the components running on a 5 Pi4 node cluster managed by K3s.

3.2 Hardware

For our edge cluster, we purchased a Picocluster 5H. The cluster consisted of 5 RPI 4B 8GB with 128 GB MicroSD cards for storage. We imaged each of the MicroSD cards with 64-bit Raspberry Pi OS Lite, specifically version 11. We additionally imaged them with the hostnames of: Master0, Master1, Master2, Worker0, and Worker1. After imaging the MicroSD Cards, we enabled them to connect to the internet via the local WiFi, and to be accessed and controlled remotely using ssh. We assembled the cluster with the combined Raspberry Pis, as well as power, cooling, and networking systems. To confirm that the hardware had been assembled correctly, we powered the system on to smoke test it. To confirm that the Raspberry Pis had been imaged and

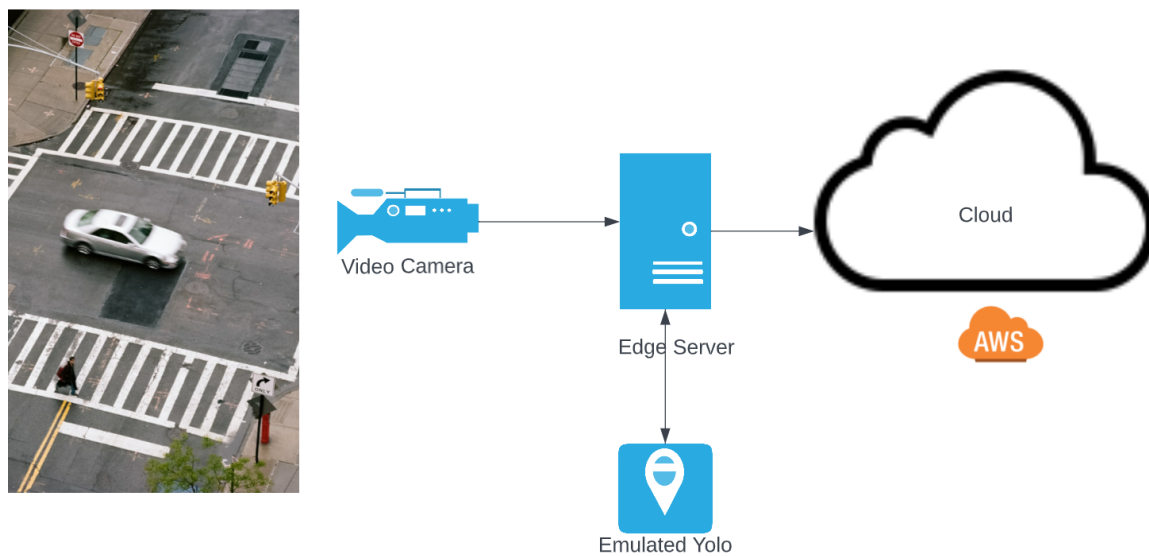


Figure 3.1: Baseline edge system

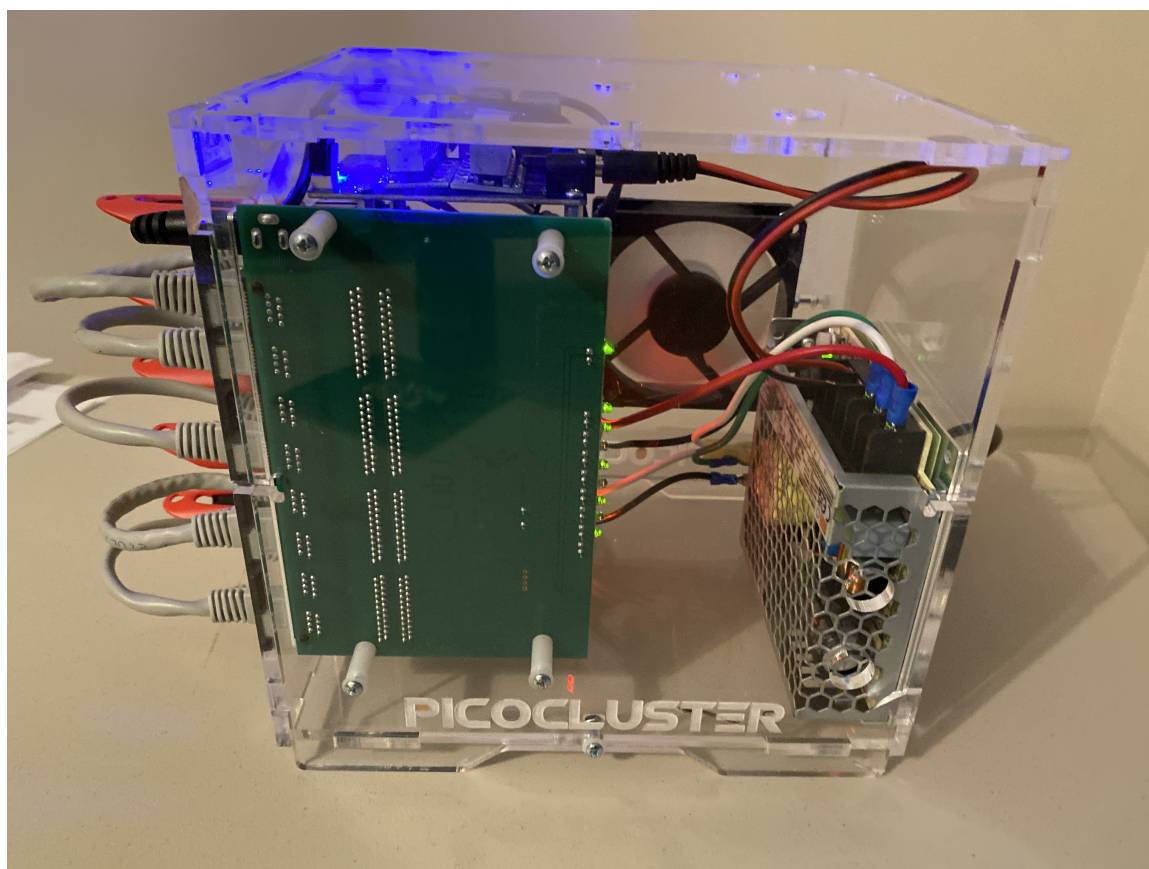
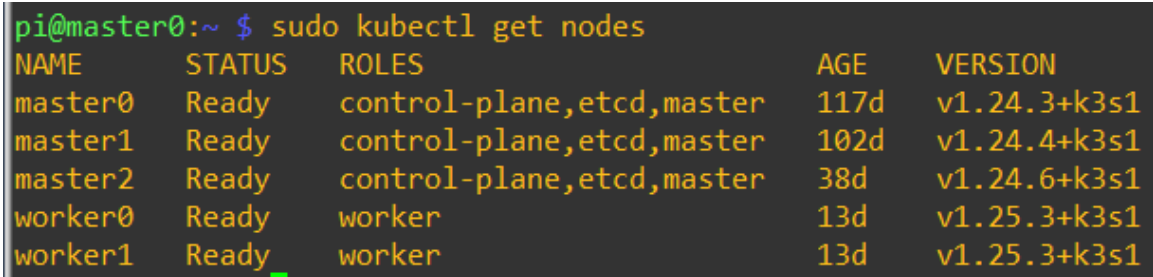


Figure 3.2: Picocluster edge testbed

configured correctly, we remotely accessed each Raspberry Pi with using ssh. Once we confirmed that each of them could be remotely accessed correctly, we were able to install K3s on each of the Raspberry Pis.

3.3 Installing K3s

We installed K3s on each of the Pis, and configured the Pis into a High Availability K3s Cluster with embedded DB. First, we installed K3s on the Master0 with the command `curl -sfL https://get.k3s.io | K3S_TOKEN=SECRET sh -s - server -cluster-init`. In order to connect other nodes to the to the cluster, we retrieved the secret token of the first node from the K3s configuration file that is generated when a K3s master node is generated in this way. This secret token is needed to connect other nodes the the cluster. To connect each of the other Master nodes to the cluster, we used the command: `curl -sfL https://get.k3s.io | $K3S_TOKEN= K3S_TOKEN sh -s - server -server https://master0:6443`. After all of the Master nodes were connected, we then connected the worker nodes with the following command: `curl -sfL https://get.k3s.io |K3S_EXEC="agent" K3S_URL=https://10.42.0.0:6443 K3S_TOKEN= $K3S_TOKEN sh -`. We verified that each node was connected to the cluster and running by getting all nodes from kubectl, as can be seen in 3.3. For further verification, we used a multiple pod Nginx deployment to check functionality for all nodes in the cluster



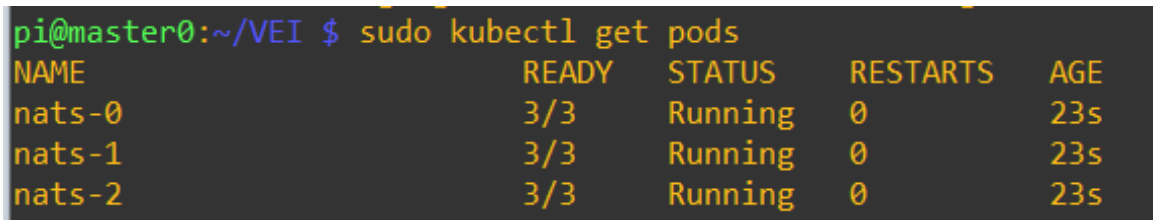
```
pi@master0:~ $ sudo kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
master0	Ready	control-plane,etcd,master	117d	v1.24.3+k3s1
master1	Ready	control-plane,etcd,master	102d	v1.24.4+k3s1
master2	Ready	control-plane,etcd,master	38d	v1.24.6+k3s1
worker0	Ready	worker	13d	v1.25.3+k3s1
worker1	Ready	worker	13d	v1.25.3+k3s1

Figure 3.3: Screenshot of Pi4 nodes running in K3s cluster

3.4 NATS in K3s

The first component of VEI that we installed on the K3s cluster was NATS. We first deployed NATS server to the K3s cluster using the NATS Helm chart with 3 replicas. Helm is the Kubernetes package manager. According to the NATS documentation, this is also known as High Availability mode. To make sure that NATS was working correctly, we used the NATS-Box pod to test the basic publish and subscribe functionality.



```
pi@master0:~/VEI $ sudo kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
nats-0        3/3     Running   0           23s
nats-1        3/3     Running   0           23s
nats-2        3/3     Running   0           23s
```

Figure 3.4: Screenshot of NATS running on K3s in High Availability mode with 3 Replicas

3.5 Containerizing Application

To run VEI and the emulated vision application on the K3s cluster, we first needed to build them into Docker images. There were a few changes that we made such that applications would function when built as Docker containers. The API for VEI was modified to look for a NATS service running in K3s. We included the AWS SDK package VEI dependencies within the container image. We passed the AWS key and security key required to access AWS IoT Core service into the container as environment variables, such as `sudo docker run -d --name server --env AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID --env AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY --network=nats-net -p 50051:50051 bobsmithy/vei-server`. Additionally, we changed the IP addresses that the different containers were looking for to ones exposed by the Docker containers. As an intermediate step, we tested the system with standalone Docker containers to confirm their functionality. When this

was confirmed, we could run the Docker images as pods and services in K3s.

3.6 Running Containers in K3s

Since the VEI gateway API uses *gRPC*, we cannot naively deploy them as K3s pods. When we attempted to do so, the standard liveness and health checks that K3s uses continually concluded that the K3s pods are unhealthy, and then restarted them. This repeats indefinitely, with the pods entering a crash loop. This is because the standard health and liveness checks that K3s uses by default use the default health and liveness probes. The default probes use http, so they cannot communicate with the application that uses *gRPC*. As such, we had to create and add *gRPC* health probes and *grpcurl* to the docker image, the code used for the probes can be found in A.1.

Additionally, we had to create and configure additional liveness and readiness probes to the YAML file for any pod or deployment for the VEI server. The code which configures the probes for a Kubernetes pod or deployment can be found in Appendix A.2. We passed the AWS key and secret key into the Kubernetes pods as environment variables in the pods' YAML file so that it could authenticate to AWS. Finally, we changed the addresses that the programs in each pod and application were looking for as to match the DNS names exposed by the K3s services. The YAML files for the server application running in K3s can be found in Appendix A.3 and Appendix A.4

```
pi@master0:~$ sudo kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED	NODE	READINESS	GATES
nats-0	3/3	Running	3 (115s ago)	14m	10.42.7.220	worker1	<none>		<none>	
nats-1	3/3	Running	0	14m	10.42.4.55	master2	<none>		<none>	
nats-2	3/3	Running	0	14m	10.42.0.155	master0	<none>		<none>	
nats-box-54ff979659-gm5gb	1/1	Running	0	14m	10.42.4.54	master2	<none>		<none>	
vei-deployment-5c4f4d7d78-247td	1/1	Running	0	12m	10.42.4.57	master2	<none>		<none>	
vei-deployment-5c4f4d7d78-h8p56	1/1	Running	0	12m	10.42.0.156	master0	<none>		<none>	
vei-deployment-5c4f4d7d78-lqmc	1/1	Running	0	12m	10.42.7.218	worker1	<none>		<none>	
yolo-deployment-6f65c567d5-6b4rt	1/1	Running	1 (22s ago)	44s	10.42.4.59	master2	<none>		<none>	
yolo-deployment-6f65c567d5-bfvfx	0/1	Error	2 (42s ago)	13m	10.42.7.221	worker1	<none>		<none>	
yolo-deployment-6f65c567d5-rhrqr	1/1	Running	1 (28s ago)	53s	10.42.3.123	master1	<none>		<none>	

Figure 3.5: All VEI Components running in K3s in High Availability mode with 3

3.7 Monitoring overhead of system in K3s

To monitor the resources consumed by the K3s cluster, we used a combination of Prometheus Operator, Prometheus, and Grafana running in K3s. Since storing the data that Prometheus scraped from the pods locally on the Raspberry Pis would consume memory, we instead stored them on removable drives - 64 GB USB drives connected to each of the Raspberry Pis. We wiped, configured, and mounted the drives. We then configured Prometheus and Grafana on K3s to store their data in the USB drives instead of on the Raspberry Pis.

Rather than install the Prometheus for Kubernetes Helm Chart to deploy Prometheus, we instead used our own custom deployments instead. This way, we could provide the same functionality in scraping, displaying, and exporting data about the Kubernetes system, but with a lower overhead. We installed Prometheus Operator for Kubernetes, though we modified it to run in a new monitoring namespace. We used a custom service monitor based on the example of Vladimir Strycek [28]. We used the service monitor for kubelet to expose the CPU and memory used by the pods.

We used a persistent Prometheus deployment which scrapes the metrics exposed by the kubelet service monitor and stores the data on a Longhorn storage volume. The data stored is available on a Prometheus webserver which can be accessed outside of the cluster. The YAML deployment can be found on Appendix A.8. With this, we were able to view the CPU and memory used by each pod in the system. However, Prometheus is not, in itself capable of exporting data to be analysed. For that, we used a Grafana deployment to visualize the data generated by Prometheus in the form of a dashboard, and to export the dashboard as a CSV of time series data [28]. We used the CSV data for offline analysis.

To better characterize the effect of a node failure on a system, we also added a boot delay of 120 seconds in config.txt file of each Pi4 board. We also wrote a brief bash script which returned the current unix timestamp, then rebooted the Pi4.

3.8 Measuring baseline system

We measured the overhead and latency of the baseline system, or the system when not running in K3s. That way, we could compare the additional resource consumption and latency imposed by running the system in K3s. The baseline system consists of three components: the edge gateway running on bare metal, the emulated vision application running on bare metal, and the NATS server running in a Docker Container.

To measure and record the overhead of the baseline programs, we instrumented both the emulated vision application and VEI server to send metrics to Prometheus. Additionally we used a service called `prometheus-nats-exporter`, which reads the information about the NATs server and exports it so that it can be read by Prometheus.

We created a standalone Prometheus service, which was configured to read the information from the different components of the baseline system. To visualize and export the metrics, we used a standalone Grafana service.

CHAPTER 4: Experimental Results

In this section, we describe the evaluation of the overhead and functionality of the VEI edge gateway, and the emulated computer vision application orchestrated by K3s as described in Chapter 3. We first characterize the resource usage (memory and CPU) of the system with K3s. Next, we measure the impact of K3s on the system latency. We then investigate the fault tolerance aspects of K3s. Our measurements are compared against performance of the baseline edge gateway implementation.

4.1 CPU and memory utilization

We first establish the CPU and memory consumed by the containerized edge gateway. The VEI gateway was run on Raspberry Pi 4 Model B boards. To obtain the resource usage, Prometheus and Grafana monitoring and visualization tools described in Section 3 were used. The services are run background daemons. The VEI gateway and the associated NATS messaging service are instrumented to export metrics to Prometheus.

We first measure the quiescent system resource usage, that is, when no video frame data is published to the gateway. VEI (including NATS) and the emulated YOLO, and the NATS server running in a Docker container were allowed to run for 30 minutes. To measure the CPU usage of the system, the Prometheus promQL query *process_cpu_seconds* was used. To measure the memory used by the system, the promQL query *process_memory_bytes* was used. After 30 minutes had elapsed, we downloaded the time series measurement data from Grafana consisting of all of the metrics generated from the queries. The query generated 4 sets of metrics: for the VEI server, the associated NATS messaging system, for the emulated computer vision

application, and for Prometheus itself. The data from Prometheus was discarded, and the metrics from the remaining values were summed up to measure the total memory and CPU used at each measurement interval. The CPU usage is measured in terms of total core usage with a maximum value of 4 (Pi4 board has a quad core processor). The memory usage is measured in Megabytes. The CPU and memory measurements are repeated with the camera client streaming video frames at a rate of 10 frames per second (fps). The average frame size is 24 KB.

Figure 4.1 shows the box plot of the system resource consumption. The median quiescent CPU usage is 0.06 cores. This increases to 0.289 CPU cores when video frames are streamed by the camera client. Regarding memory, the median quiescent memory usage is 110.8 MB. This increases to 111.3 MB when video frames are streamed by the camera client. Note that the Pi4 board has a total of 8 GB of memory. From the results we see that the system is adequately provisioned in terms of CPU and memory, for running the VEI edge server, and an emulated vision application.

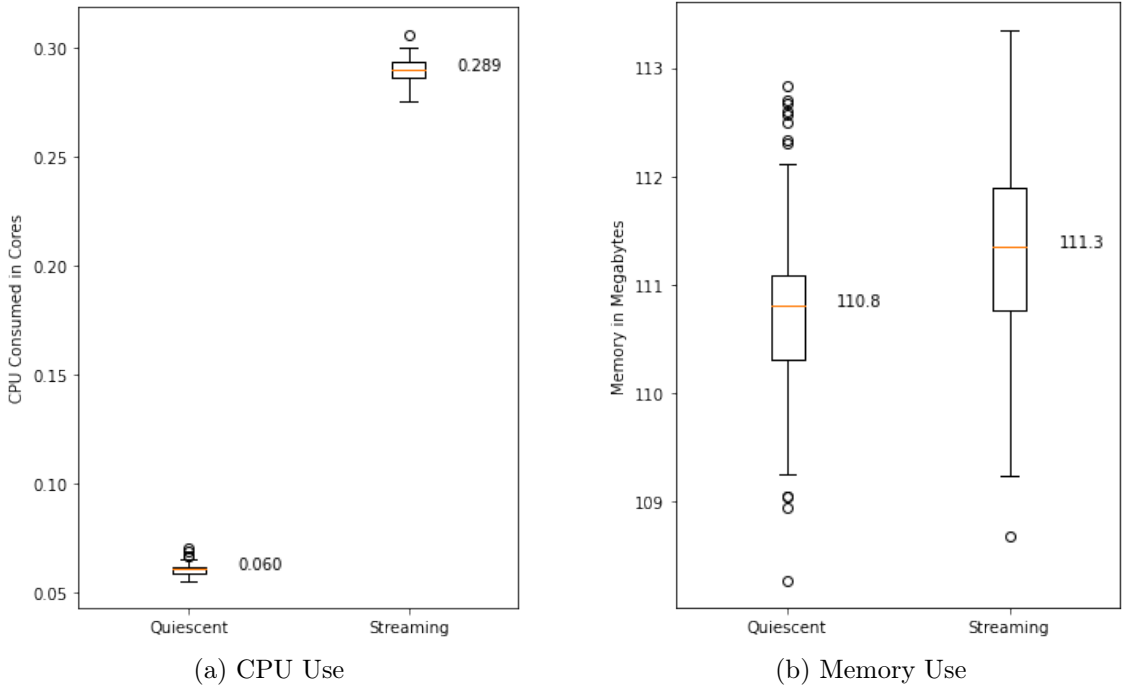


Figure 4.1: Baseline System Resource Consumption

Next we establish the CPU and memory consumed by the Edge Gateway when running on a K3s cluster. The K3s cluster was run on 5 Raspberry Pi 4 Model B boards, with 3 Master Nodes and 2 Worker Nodes. The Edge Gateway in K3s consists of the containerized VEI server, the containerized emulated vision application, and the containerized NATS server. Each component is deployed with three copies, each of which run on a different node on the cluster. To obtain the resource usage of the pods within the cluster, we used the Prometheus, and Grafana K3s deployments as described in Chapter 3.

First, we measured the resource usage of the system when it was quiescent, that is, when no data was being streamed to it by the camera. The system in K3s was permitted to run for 30 minutes. To measure the CPU usage of the system, the Prometheus promQL query *process_cpu_seconds by pod_name* was used. To measure the memory used by the system, the promQL query *process_memory_bytes by pod_name* was used. After 30 minutes had elapsed, we downloaded the time series measurement data from Grafana consisting of the metrics generated from the queries for each of the pods in the Edge Gateway (9 in total). For each measurement interval, we summed the values of all the pods together to get the total CPU and memory used at that measurement interval by the Edge Gateway as a whole. The CPU usage is measured in terms of total core usage with a maximum value of 20 (4 cores for each Pi 4 board, with 5 boards in the cluster). The memory usage is measured in Megabytes. The CPU and memory measurements are then repeated with the the camera client streaming video frames at a rate of 10 frames per second (fps). The average frame size is 24 kB.

Figure 4.2 shows the box plot of the system resource consumption. When the system is in a quiescent state, it has a median consumption 0.128 cores of CPU. However, when it is being streamed images, it increases to a median consumption of 7.51 cores of CPU. With respect to memory, the median quiescent usage is 125.5 MB.

When the system is streamed images, it consumes 213.4 MB of memory. From this we can see that the resource consumption of an active K3s cluster is significantly more than than the quiescent state. We discuss this point in further detail in Chapter 5.

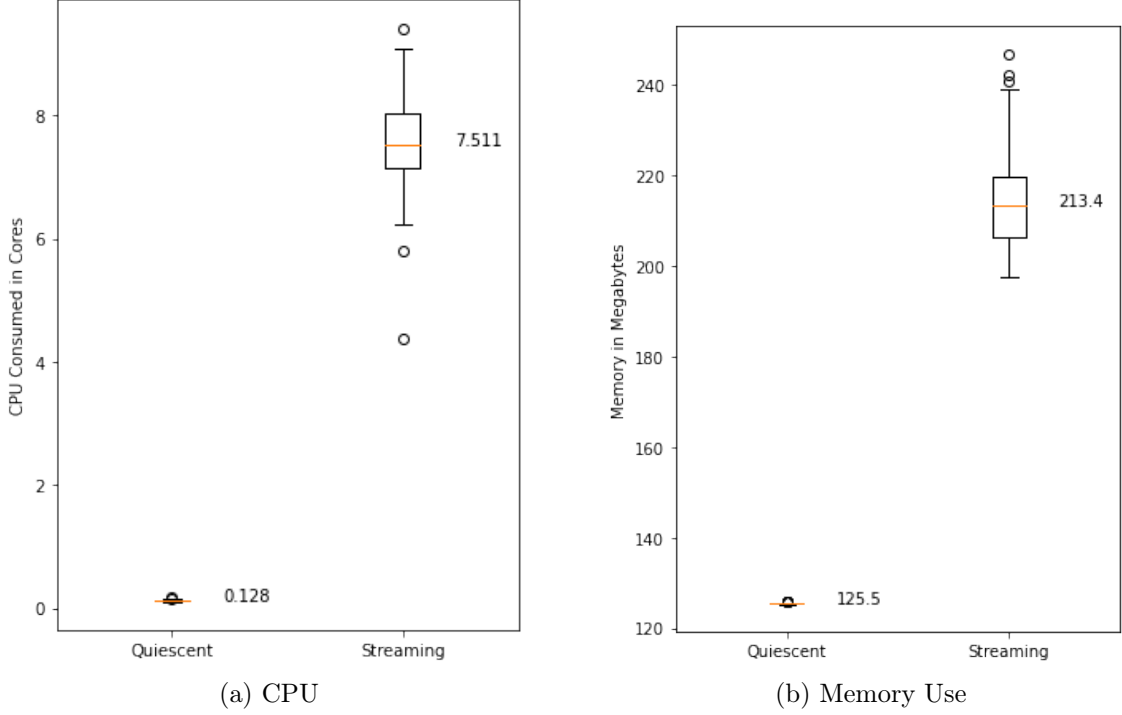


Figure 4.2: Resource consumption of system in K3s

4.2 Latency

We next measure the impact of K3s on the overall latency of the gateway and vision application. To measure the latency of the system, the camera application and emulated vision application were modified to generate timestamps. The first timestamp (the creation timestamp) was generated by the camera client immediately before it sends the video frame to the system, and the second timestamp (the system timestamp) was generated before the data is published to the AWS cloud. The difference between the two timestamps represents the end-to-end system latency.

We first measure the latency of the baseline system. The system, consisting of VEI (including NATS) and the emulated vision application, and the NATS server running in a Docker container is streamed images by the camera client. The server

was streamed 1000 frames at a rate of 10 frames per second. When the 1000 frames streaming had elapsed, the timestamps were retrieved from the AWS cloud.

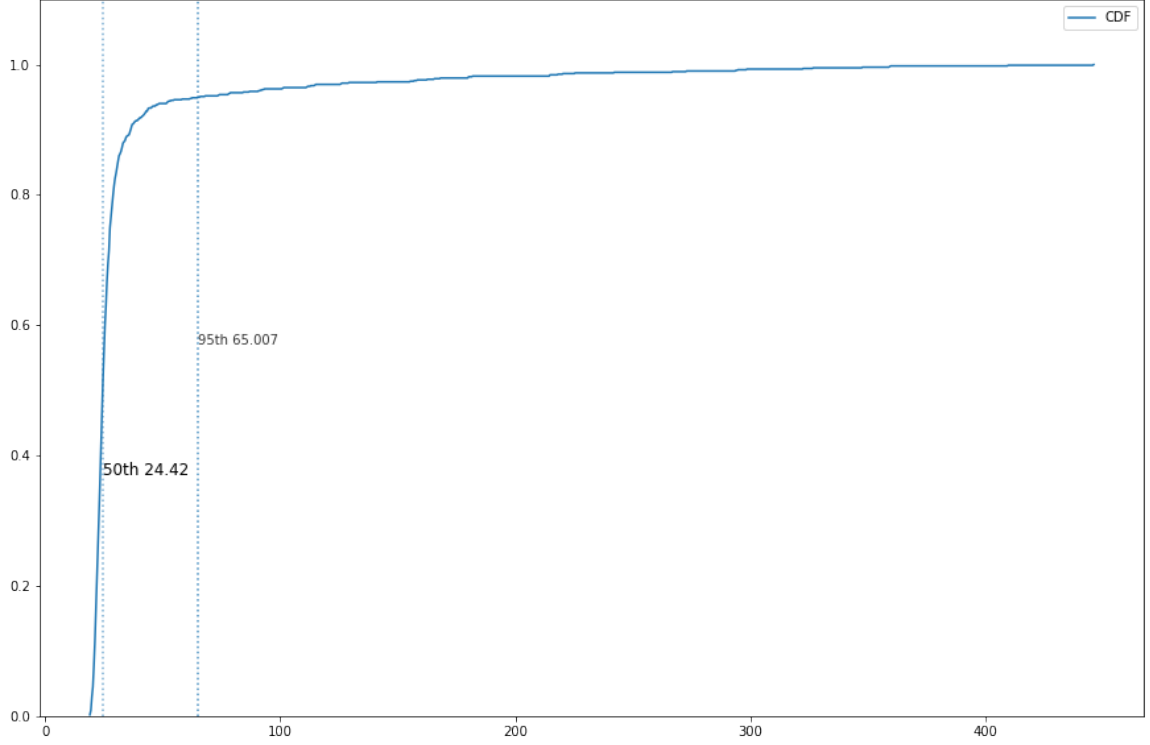


Figure 4.3: Cumulative Distribution Function (CDF) of latency (milliseconds) for baseline edge system. The latency CDF was plotted over 1000 frames at a rate of 10 FPS

Figure 4.3 shows the Latency Cumulative Distribution Function (CDF) of the system for 1000 images. The median Latency is 24.420 ms, and the 95th percentile latency is 65.007 ms. From the results we see that the system has an adequate latency for real time edge vision applications.

Next, we establish the latency of the system when orchestrated by K3s. The K3s cluster was run on 5 Raspberry Pi 4 Model B boards, with 3 master Nodes and 2 worker Nodes. The Edge Gateway in K3s consists of the containerized VEI server, the containerized emulated vision application, and the containerized NATS Server. Each of components is deployed with three copies, each of which run on a different node on the cluster. To compute the latency of the system, two timestamps are collected by

the system. One is created by the camera application when it sends a frame to the system. The second is created immediately before the application uploads the data to the AWS cloud.

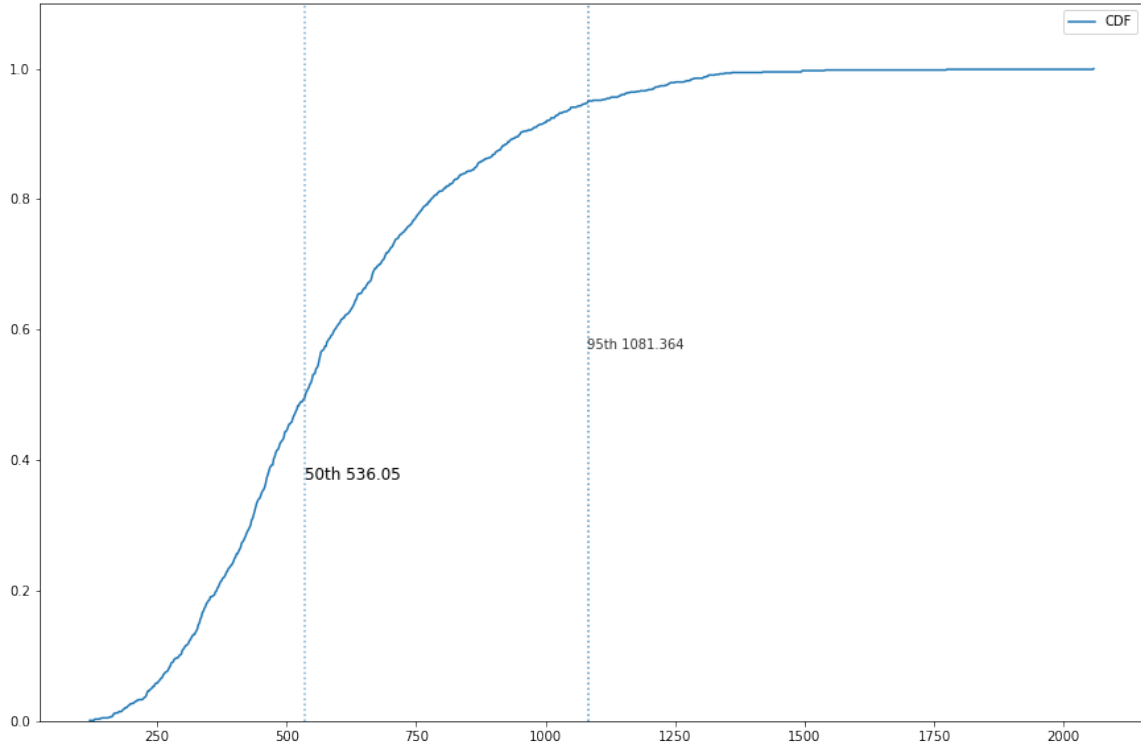


Figure 4.4: Cumulative Distribution Function (CDF) of VEI latency (milliseconds) for edge system in K3s. The latency CDF was plotted over 1000 frames at a rate of 10 FPS

The system was streamed 1000 images by the camera client at a rate of 10 fps. When the 1000 frames had elapsed, the data was retrieved from the cloud. As before, the difference between the two timestamps represents the end-to-end system latency.

Figure 4.4 shows the latency Cumulative Distribution Function (CDF) of the system in K3s for 1000 images. The median Latency is 536.050 ms, and the 95th percentile latency is shown to be 1081.364 ms. From the results we see that the system has a significantly higher latency in K3s as compared to the baseline.

4.3 Characterization of Node Failure

Finally, we establish the latency impact of recovery on node failure of the K3s system to different single worker node failure scenarios. The K3s cluster was run on 5 Raspberry Pi 4 Model B boards, with 3 master Nodes and 2 worker Nodes. The edge gateway in K3s consists of the containerized VEI server, the containerized emulated vision application, and the containerized NATS Server. Each of components is deployed with three copies, and each copy runs on a different node on the cluster. To compute the latency of the system, two timestamps are collected by the system. One is created by the camera application when it sends a frame to the system. The second is created immediately before the application uploads the data to the AWS cloud.

All failure scenarios are studied is the same way. The system is first streamed images from the camera application at a rate of 10 frames per second. At a marked time, a node with of the desired containers running on it is terminated manually. When the system had recovered, all of the data is retrieved from the Cloud.

We first fail a node that runs the NATS messaging service by rebooting it with a delay of 2 minutes. When the system recovers, all of the data is retrieved from the cloud.

Figure 4.5: Failure of Node with only NATS

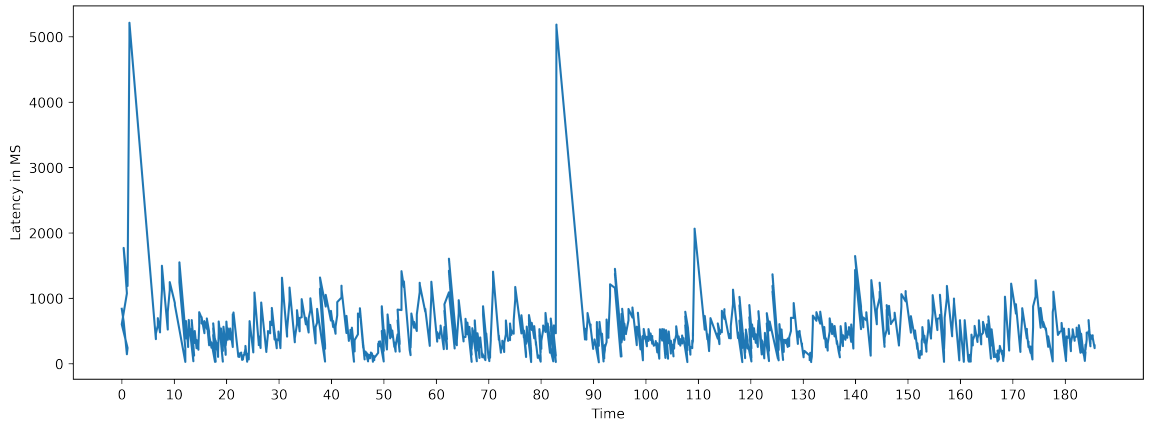


Figure 4.5 shows the response of the system from the time when the node is terminated to when the system recovers to the baseline latency. K3s switches the NATS service to a replica running on a functional node. After the failure, several spikes in latency are observed. From this we can see that the system in K3s has acceptable fault tolerance for this failure.

We next characterize the response of the system to the failure of a node with only the emulated computer vision application.

Figure 4.6: Failure of node with emulated vision application

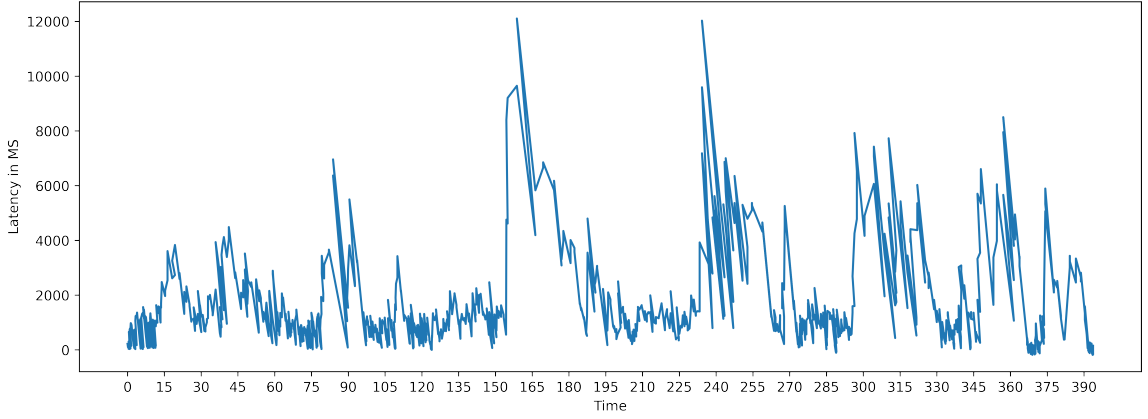
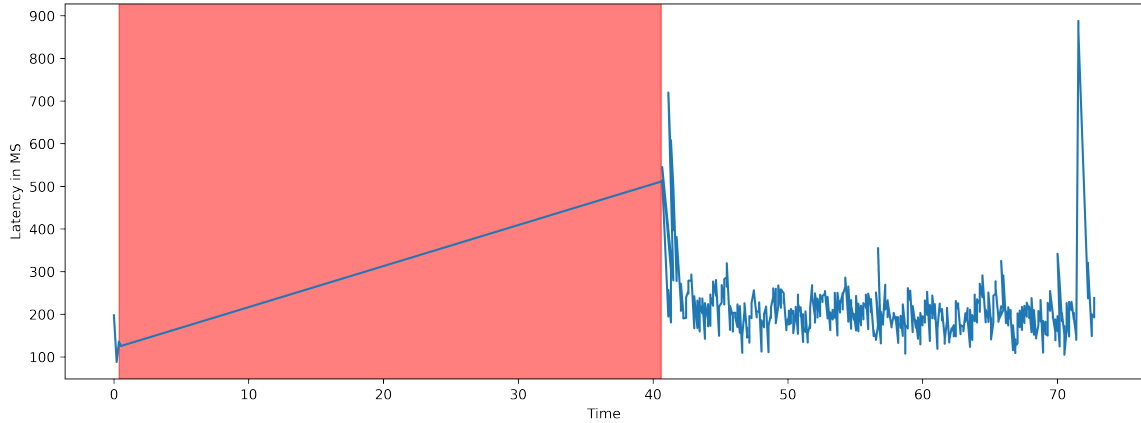


Figure 4.6 shows the response of the system to the failure of a node with only a replica of the emulated computer vision application running on it, from when the node is terminated to when the system recovers to a normal latency. The recovery from the failure is characterized by higher latency. From this we can see that the system in k3s has acceptable fault tolerance for this failure.

We next characterize the response of the system to the failure of a node with only the VEI edge gateway running on it.

Figure 4.7 shows the response of the system to the failure of a node with only a replica of the VEI edge gateway running on it, from when the node is terminated to when the system recovers to a normal latency. The period of system failure is marked in red. In this period (lasting approximately 40 seconds), the cloud received no data

Figure 4.7: Failure of node with VEI edge gateway



from the edge gateway. From this we can see that despite K3s offering fault tolerance and system recovery, there are still large system outages in the event of some node failures.

Next, we characterize the tolerance of the system to a node failure with a copy of the VEI Server and a copy of the emulated vision application running on it.

Figure 4.8: Failure of node with VEI and emulated vision application

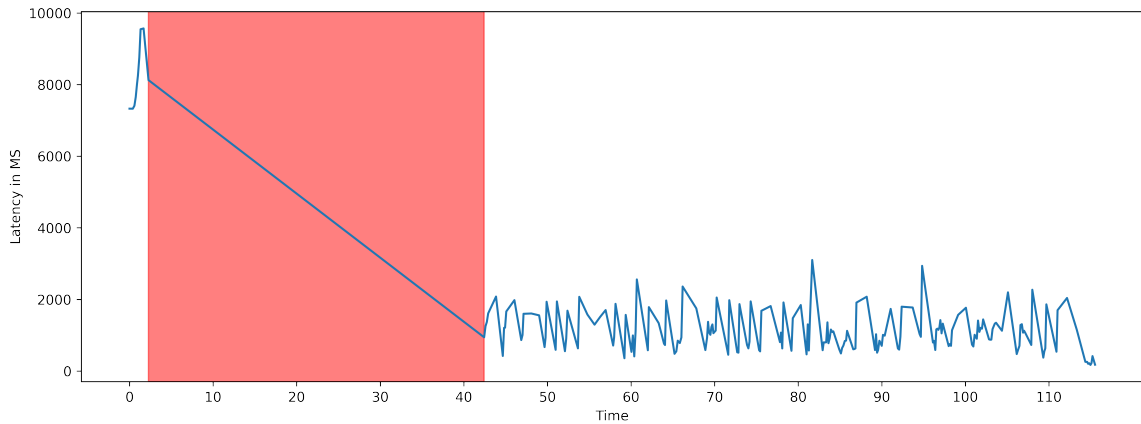


Figure 4.8 shows the response of the system when a node with a replica of the VEI edge gateway, and a replica of the emulated vision application running on it fail. When the node fails there is a system failure for approximately 40 seconds (Marked in Red). This is followed by a period of extremely high average latency which lasts

for several minutes. From this we can conclude that there despite K3s offering some fault tolerance, there are still system outages, and a long period of extremely high latency.

We then characterize the system response to the failure of a node with both NATS server and the emulated vision application running on it.

Figure 4.9: Failure of node with NATS and the emulated vision application

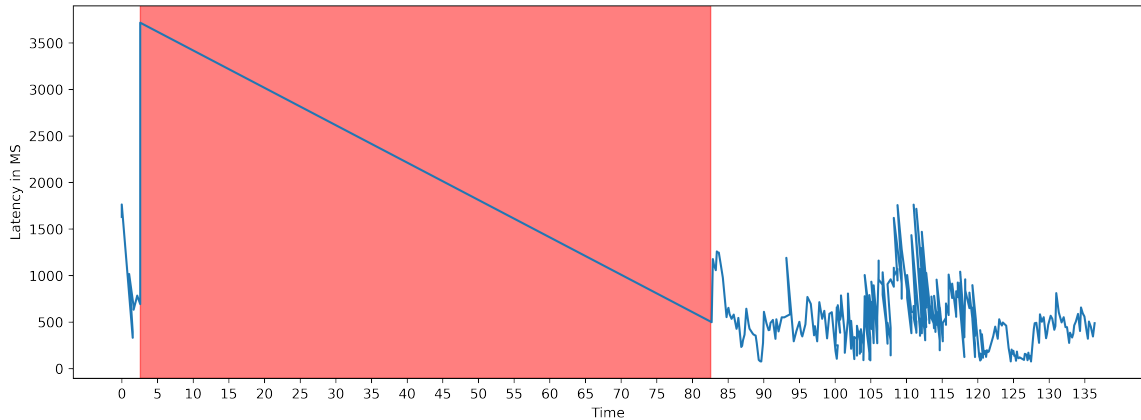
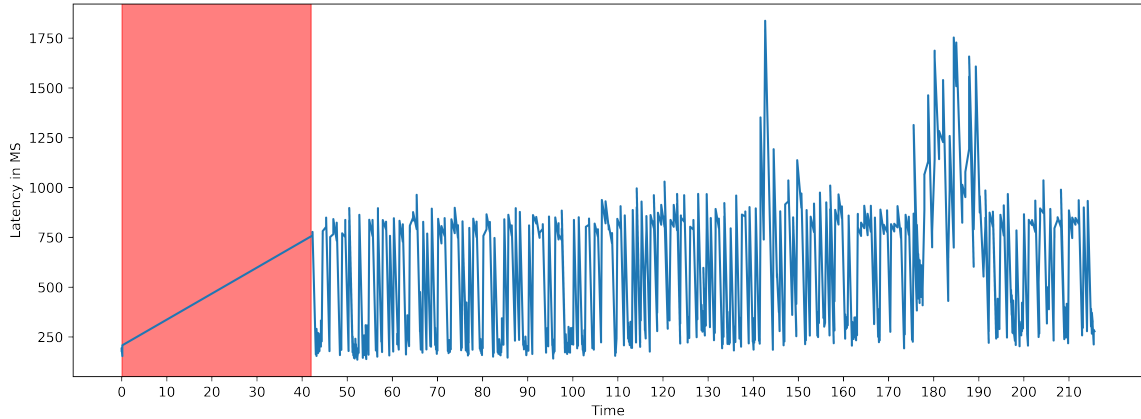


Figure 4.9 shows the response of the system when a node with a replica of the NATS Server, and a replica of the emulated of the vision application running on it fails. When the node fails there is a failure on of the system for 80 seconds (Marked in Red). This is followed by a period of higher latency for several minutes. From this we can conclude that though K3s offers some fault tolerance, there are still large system outages which last for more than a minute.

Next, we characterize the system response to the failure of a node with both NATS server and the emulated vision application running on it.

Figure 4.10 shows the response of the system when a node with a replica of VEI, and the a copy of the emulated vision application running on it fail. When the node fails there is an outage on of the system for 40 seconds (Marked in Red). This outage is followed by a period of baseline latency, punctuated by several spikes in latency before the system recovers. From this we can see that even though K3s offers fault

Figure 4.10: Failure of node with NATS and VEI



tolerance, there are still significant system outages when a node with a replica of VEI and a replica of the emulated vision application running on it fails.

Next, we characterize the system response to a node failure with all three deployment replicas: NATS Server, VEI Server, and the emulated vision application running on the node.

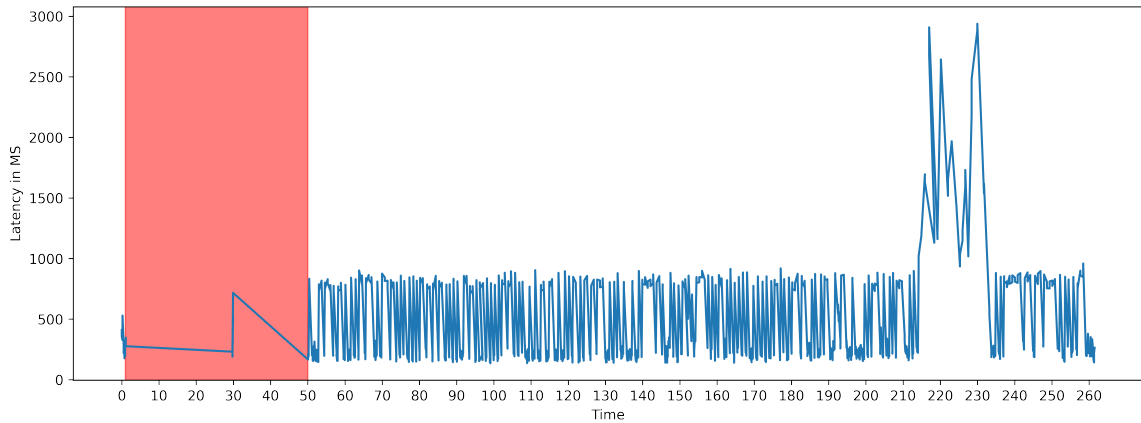


Figure 4.11: Characterization of failure of node with NATS, VEI, and the emulated vision application running

Figure 4.11 shows the response of the system when a node with VEI, NATS, and the emulated vision application running on it fails. When the node fails there is a failure of the system for 40 seconds (Marked in Red). This is followed by a period of high latency for several minutes. When the node rejoined the cluster, and the pods

restarted, there was a further spike of high latency, before it returned to the high latency. After several seconds, the system then returned to normal. From this we can see that even though K3s offers some fault tolerance, there is still a period of failure and high latency when this node fails.

CHAPTER 5: Discussion

In this section, we place our work in the larger context of solutions to the problem of fault tolerance at the edge. Our application of K3s to an edge gateway was motivated by the need to provide fault tolerance for edge devices running complex applications.

Running the system in K3s increased the median latency of the system by 2095.13%, and increases 95th percentile latency by 1563.46%. In terms of CPU cost, 72.3404% when quiescent, increases by 185.18% when being streamed images at 10 frames per second. In terms of memory 12.442 % when quiescent, and 62.889 % when being streamed images at 10 frames per second. For node failures, we found that for 5 out of 7 different single node failures, there were system outages lasting at least 40 seconds.

These results show that K3s is a viable solution for fault tolerance on the edge, as the system does recover in all situations that we tested. However, K3s on the Raspberry Pi cluster is not necessarily suited for every application. When the system is quiescent, it consumes a median value 0.128 cores of CPU. However, when the system is being streamed images from the camera client, it consumes a median value of 7.511 cores of CPU, which is a massive increase. This value is especially large when you consider that no image processing is occurring on the edge device.

However, there are many applications to which the K3s on a Pi4 cluster is a viable solution. These include where latency requirements are fairly relaxed, the object detection algorithms run directly on the camera, and the application running on the edge server act on detected/tracked objects.

We see three possible directions for future research. First, it could be possible that the source of the high latency and resource consumption of the system in K3s

is a result of the system design. For instance, it is possible that communications between the pods are the primary source of increase in resource consumption. If that is the case, it is possible that combining each of the components into a single pod and replicated it across the cluster could be a superior solution. Second, it is possible that Raspberry Pis are too low powered for running applications on K3s with low latency. Another system, with more CPU, or an embedded GPU, such as a Jetson Xavier, might be able to use applications in K3s with more success [18]. A third direction is to explore alternatives to K3s such as Nomad scheduler and orchestrator from Hashicorp [15]. Nomad however does not provide as much functionality as Kubernetes does. An alternate approach would also be to design an edge specific system from scratch such as the recently published FLEDGE container orchestrator [14].

CHAPTER 6: Conclusion

In this chapter, we summarize our work.

In this thesis, we have investigated K3s on a cluster consisting of low power ARM based Raspberry Pi 4 nodes as a solution for fault tolerance on the edge. We noted that existing research on the resource consumption of container orchestration has not been characterized on low power edge platforms. We experimentally measured consumption of CPU and memory by a system on K3s compared to baseline, characterized the added application latency due to K3s, and evaluated the latency impact of recovery on node failure under multiple failure scenarios. Our results show that while K3s does impose a latency penalty, it is a viable solution to the problem of fault tolerance at the edge. We also noted several future extensions of our work.

REFERENCES

- [1] AWS University. What is IoT. Accessed Dec. 4, 2022 [Online].
- [2] Banerjee, Sayanta. Everything you need to know about k3s: Lightweight kubernetes for IOT, Edge Computing, Embedded Systems and more. Accessed Dec. 4, 2022 [Online].
- [3] Sebastian Böhm and Guido Wirtz. Towards Orchestration of Cloud-Edge Architectures with Kubernetes. In *Science and Technologies for Smart Cities*, 11 2021.
- [4] Keyan Cao, Yefan Liu, Gongjie Meng, and Qimeng Sun. An overview on edge computing research. *IEEE Access*, 8:85714–85728, 2020.
- [5] Hamed Chourabi, Taewoo Nam, Shawn Walker, J Ramon Gil-Garcia, Sehl Mellouli, Karine Nahon, Theresa A Pardo, and Hans Jochen Scholl. Understanding smart cities: An integrative framework. In *2012 45th Hawaii international conference on system sciences*, pages 2289–2297. IEEE, 2012.
- [6] Clark, Jen. What is the internet of things, and how does it work? Accessed Dec. 4, 2022 [Online].
- [7] Docker. Docker Overview.
- [8] Andreas Eiermann, Mathias Renner, Marcel Großmann, and Udo R Krieger. On a fog computing platform built on arm architectures by docker container technology. In *International Conference on Innovations for Community Services*, pages 71–86. Springer, 2017.
- [9] Ellingwood, Justin. An introduction to kubernetes.
- [10] Halim Fathoni, Chao-Tung Yang, Chih-Hung Chang, and Chin-Yin Huang. Performance comparison of lightweight kubernetes in edge devices. In Christian Esposito, Jiman Hong, and Kim-Kwang Raymond Choo, editors, *Pervasive Systems, Algorithms and Networks*, pages 304–309, Cham, 2019. Springer International Publishing.
- [11] Xin Feng, Youni Jiang, Xuejiao Yang, Ming Du, and Xin Li. Computer vision algorithms and hardware implementations: A survey. *Integration*, 69:309–320, 2019.
- [12] David Forsyth and Jean Ponce. *Computer Vision: A Modern Approach. (Second edition)*. Prentice Hall, November 2011.

- [13] FOSS TechNix. Kubernetes Concepts for beginners [8 k8s components]. Accessed Dec. 4, 2022 [Online].
- [14] Tom Goethals, Filip De Turck, and Bruno Volckaert. *FLEDGE: Kubernetes Compatible Container Orchestration on Low-Resource Edge Devices*, pages 174–189. 01 2020.
- [15] Hashicorp. Introduction to Nomad . Accessed Dec. 4, 2022 [Online].
- [16] Yi-Zeng Hsieh, Fu-Xiong Xu, and Shih-Syun Lin. Deep convolutional generative adversarial network for inverse kinematics of self-assembly robotic arm based on the depth sensor. *IEEE Sensors Journal*, pages 1–1, 2022.
- [17] Hussein Galal. Introduction to k3s. Accessed Dec. 4, 2022 [Online].
- [18] Jetson. Jetson AGX Xavier Series Modules . Accessed Dec. 4, 2022 [Online].
- [19] Peiyuan Jiang, Daji Ergu, Fangyao Liu, Ying Cai, and Bo Ma. A review of yolo algorithm developments. *Procedia Computer Science*, 199:1066–1073, 2022. The 8th International Conference on Information Technology and Quantitative Management (ITQM 2020 and 2021): Developing Global Digital Economy after COVID-19.
- [20] Wazir Zada Khan, Ejaz Ahmed, Saqib Hakak, Ibrar Yaqoob, and Arif Ahmed. Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235, 2019.
- [21] Kuberenetes. Overview. Accessed Dec. 4, 2022 [Online].
- [22] Arseni Leskinen. Applicability of Kubernetes to Industrial IoT Edge Computing System. Master’s thesis, Aalto University. School of Electrical Engineering, 2020.
- [23] Samantha Luu, Arun Ravindran, Armin Danesh Pazho, and Hamed Tabkhi. Vei: A multicloud edge gateway for computer vision in iot. In *Proceedings of the 1st Workshop on Middleware for the Edge*, MIDDLEWEDGE ’22, page 6/11, New York, NY, USA, 2022. Association for Computing Machinery.
- [24] NATS. What is NATS - NATS Docs. Accessed Dec. 4, 2022 [Online].
- [25] Patel, Ashish. Kubernetes-architecture overview. Accessed Dec. 4, 2022 [Online].
- [26] Rancher. What is k3s. Accessed Dec. 4, 2022 [Online].
- [27] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.
- [28] Strycek, Vladimir. Montioring a Raspberry Pi 4 k3s Kuberentes Cluster. Accessed Dec. 4, 2022 [Online].

APPENDIX A: Configuration Scripts for K3s

A.1 GRPC healthcheck Dockerfile

Listing A.1: Code for Configuring GRPC Healthchecks for the in Dockerfile

```
FROM golang:1.16 AS grpc-health-probe-builder
RUN GRPC_HEALTH_PROBE_VERSION=v0.3.6 && \
    wget -qO/bin/grpc_health_probe https://github.com/grpc-ecosystem/grpc-health-probe/releases/download/${GRPC_HEALTH_PROBE_VERSION}/grpc_health_probe-linux-amd64 && \
    chmod +x /bin/grpc_health_probe

FROM golang:1.16 AS grpcurl-builder
RUN go get github.com/fullstorydev/grpcurl/...
RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go install github.com/fullstorydev/grpcurl/cmd/grpcurl@latest
```

A.2 GRPC Healthchecks in K3s

Listing A.2: Code to configure GRPC Healthchecks in the K3s Deployment

```
ports:
- name: grpc
  containerPort: 50051

livenessProbe:
  exec:
```

```

      command:
        - grpcurl
        - --plaintext
        - localhost:50051
        - ping.Pinger/Ping
    readinessProbe:
      exec:
        command:
          - grpc_health_probe
          - --addr=:50051

```

A.3 K3s YAML

Listing A.3: Deployment of VEI server in K3s

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: vei-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: server-pod
  template:
    metadata:
      labels:
        app: server-pod

```

```

spec:
  containers:
    - name: server-pod
      image: bobsmithy/vei-server:latest
      env:
        - name: AWS_ACCESS_KEY_ID
          value:
        - name: AWS_SECRET_ACCESS_KEY
          value:
      ports:
        - name: grpc
          containerPort: 50051
      livenessProbe:
        exec:
          command:
            - grpcurl
            - -plaintext
            - localhost:50051
            - ping.Pinger/Ping
      readinessProbe:
        exec:
          command:
            - grpc_health_probe
            - --addr=:50051

```

A.4 VEI Service YAML

Listing A.4: Service which exposes VEI server

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    konghq.com/protocol: grpc
  name: server-pod
  labels:
    app: server-pod
spec:
  type: LoadBalancer
  ports:
    - name: grpc
      port: 50051
      targetPort: 50051
  selector:
    app: server-pod
```

A.5 Emulated Vision Application YAML

Listing A.5: .yaml file that deploys Subscriber Client in k3s

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: yolo-deployment
spec:
  replicas: 3
```

```

selector:
  matchLabels:
    app: yolo-pod
template:
  metadata:
    labels:
      app: yolo-pod
  spec:
    containers:
      - name: yolo-pod
        image: bobsmithy/vei-yolo4:latest
        ports:
          - name: grpc
            containerPort: 50051

```

A.6 Reboot Bash Script

Listing A.6: Bash Script that prints timestamp, then reboots

```

#!/bin/bash

# Define a timestamp function
timestamp() {
  date +%s%N # current time
}

timestamp # print timestamp
reboot -f

```

A.7 Monitoring system YAML

Listing A.7: .yaml file for kubelet monitor

```

apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app.kubernetes.io/name: kubelet
    name: kubelet
  name: kubelet
  namespace: monitoring
spec:
  endpoints:
    - bearerTokenFile: /var/run/secrets/kubernetes.io/
      serviceaccount/token
      honorLabels: true
      interval: 15s
      port: https-metrics
      relabelings:
        - sourceLabels:
            - __metrics_path__
          targetLabel: metrics_path
      scheme: https
      tlsConfig:
        insecureSkipVerify: true
    - bearerTokenFile: /var/run/secrets/kubernetes.io/

```

```

    serviceaccount/token
honorLabels: true
honorTimestamps: false
interval: 15s
relabelings:
- sourceLabels:
  - __metrics_path__
    targetLabel: metrics_path
scheme: https
tlsConfig:
  insecureSkipVerify: true
- bearerTokenFile: /var/run/secrets/kubernetes.io/
  serviceaccount/token
honorLabels: true
interval: 15s
path: /metrics/probes
port: https-metrics
relabelings:
- sourceLabels:
  - __metrics_path__
    targetLabel: metrics_path
scheme: https
tlsConfig:
  insecureSkipVerify: true
jobLabel: k8s-app
namespaceSelector:
  matchNames:

```

```

    - kube-system
selector:
  matchLabels:
    k8s-app: kubelet

```

A.8 Prometheus YAML

Listing A.8: .yaml for Prometheus Persistent

```

apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus-persistent
  namespace: monitoring
spec:
  replicas: 1
  retention: 14d
  resources:
    requests:
      memory: 400Mi
  nodeSelector:
    node-type: worker
  securityContext:
    fsGroup: 2000
    runAsNonRoot: true
    runAsUser: 1000
  serviceAccountName: prometheus
  serviceMonitorSelector:
    matchExpressions:

```

```

    - key: name
      operator: In
      values:
        - kubelet
storage:
  volumeClaimTemplate:
    spec:
      accessModes:
        - ReadWriteOnce
      storageClassName: longhorn
      resources:
        requests:
          storage: 30Gi

```

A.9 Instrumentation for VEI in Golang

Listing A.9: VEI Server instrumented for Prometheus

```

func handler() {

    http.Handle("/metrics", promhttp.Handler())
    http.ListenAndServe(":2113", nil)
}

func main() {
    go handler()
    //Start listening to tcp port, if cannot connect then
    throw an error

    listen, err := net.Listen("tcp", port)
    if err != nil {

```

```

        log.Fatalf("failed to listen: %v", err)
    }

    //start the new server with grpc
    s := grpc.NewServer()
    VEIv1_0.RegisterVEIv1_0Server(s, &server{})

    // Connect to cloud service providers
    iotCore = connectToAWSIoT()
    log.Println("Connected to AWS")
    //mqttCli, topic = connectToGCPIoT()

    //log.Println("Connected to GCP")

    if err := s.Serve(listen); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}

```

A.10 Instrumentation for Emulated Vision Application in Python

Listing A.10: Emulated Vision Application instrumented for Prometheus

```

def main():

    prom.start_http_server(2114)

    frameNum = 0
    #host = '192.168.0.103'

```

```

host = 'localhost '
server_port = 50051

#instantiate a channel
channel = grpc.insecure_channel(
    '{}:{}'.format(host , server_port)
)

# bind the client and the server
stub = VEI_grpc.VEIv1_0Stub(channel)

#make a new request to specific subject
im_data_req = VEI.SubImageParams(cameraID="camera1 ")

```

A.11 Configuration File for Standalone Prometheus

Listing A.11: Prometheus Systemd Application .yaml

```

# my global config
global:
  scrape_interval: 15s
  evaluation_interval: 15s
# Alertmanager configuration
alerting:
  alertmanagers:
    - static_configs:
      - targets:
        # - alertmanager:9093

```

```

rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"
scrape_configs:
  - job_name: "prometheus"
static_configs:
  - targets: ["localhost:9090"]
- job_name: go_blank
  scrape_interval: 1s
  static_configs:
    - targets:
      - localhost:2112
- job_name: server.go
  scrape_interval: 1s
  static_configs:
    - targets:
      - localhost:2113
- job_name: yolo.py
  scrape_interval: 1s
  static_configs:
    - targets:
      - localhost:2114
- job_name: NATS
  scrape_interval: 1s
  static_configs:
    - targets:
      - localhost:7777

```