2024

# Characterizing a web server deployed in a lightweight Kubernetes cluster in an IoT computing environment

Jackson Lawrence

# CHARACTERIZING A WEB SERVER DEPLOYED IN A LIGHTWEIGHT KUBERNETES CLUSTER IN AN IOT COMPUTING ENVIRONMENT

**Jackson Lawrence**

**A THESIS**

**Submitted in partial fulfillment of the requirements**
**for the degree of Master of Science in Engineering**
**in**
**The Department of Electrical and Computer Engineering**
**to**
**The Graduate School**
**of**
**The University of Alabama in Huntsville**

**August 2024**

**Approved by:**

Dr. David Coe, Research Advisor/Committee Chair
Dr. Aleksandar Milenkovic, Committee Member
Dr. Earl Wells, Committee Member
Dr. Aleksandar Milenkovic, Department Chair
Dr. Shankar Mahalingam, College Dean
Dr. Jon Hakkila, Graduate Dean

## Abstract

# CHARACTERIZING A WEB SERVER DEPLOYED IN A LIGHTWEIGHT KUBERNETES CLUSTER IN AN IOT COMPUTING ENVIRONMENT

**Jackson Lawrence**

**A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering**

**Electrical and Computer Engineering**

**The University of Alabama in Huntsville**

**August 2024**

As the world's reliance on the internet continues to grow, so does the need for cloud-based computing solutions. These solutions offer a degree of flexibility, scalability, and reliability that traditional computing methods cannot provide. This thesis focuses on characterizing a web server deployed in a lightweight Kubernetes cluster hosted across three Raspberry Pi 4 minicomputers. Characterization metrics include the transfer rate of data from the web server as well as the CPU utilization and power consumption of the Raspberry Pi devices when under load. The background on cloud-based technology is introduced and the tools used in the thesis research are discussed. Experimental methods are then detailed and the results for several cluster-based deployments of the web server are presented. Finally, these results are compared to those obtained for an identical web server hosted locally on a Raspberry Pi device external to the Kubernetes cluster.

## Acknowledgements

I would first like to thank Dr. David Coe for his help in shaping this thesis and in providing me with the resources needed to do the work. This would not have been possible without his guidance.

I would also like to thank Dr. Aleksandar Milenkovic for encouraging me to pursue the thesis route for my Master's degree, Dr. Earl Wells for his generosity in sitting on my committee, and the ECE department as a whole for employing me as a teaching assistant for the last two years.

I would like to thank my parents and grandparents for their encouragement and continued belief in me throughout my undergraduate and graduate studies.

Lastly, I would like to dedicate this thesis to my wife, Amy. Without her tireless support and sacrifice on my behalf, I would not have been able to pursue this degree and this thesis would never have been written.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1. Introduction

## 1.1 Cloud Computing Background

Cloud computing is defined as the use of computing resources such as data processing, data storage, server hosting, etc. over the internet [1]. It is different from more traditional forms of computing in that the end user is not required to be responsible for managing the resources they use. Very few technologies have seen the explosive growth and widespread adoption that cloud computing has seen in the past decade. In 2015 the total market share for cloud-based services was 16.8 billion USD [2]. By 2022 that market size had increased to 388.47 billion USD [3]. Cloud computing is projected to have an annual growth rate of 18.9% through the year 2031, when its market size is estimated to be at 1,844.99 billion USD [3]. This rapid uptake of cloud technologies has permeated every area involving technology. Today more than 60% of corporate data are stored in the cloud and more than 2.3 billion people were estimated to use personal cloud services like Google Drive, iCloud, and Dropbox in 2020 [4] [5]. Even the majority of entertainment is now consumed via cloud-based platforms and technologies. In 2014 27.3% of annual music revenue in the United States came from streaming services [6]. By the year 2022 83.9% of revenue was from music streaming, all enabled by cloud technology [6]. The proliferation of generative AI technology is

also due, in large part, to cloud computing, as the amount of data and processing power required to support AI infrastructure is enabled by the cloud [7].

## 1.2 Cloud Architecture

At its core, cloud technology separates the management of computing resources from the end users of those resources. For much of the history of computing, individuals or businesses owned and managed the hardware they used to support their websites, networks, and other computer-based applications and resources. With cloud computing, however, much of this resource management is taken on by a third party that provides the necessary resources for the end user via the internet. Figure 1.1 illustrates a basic diagram of front-end and back-end cloud architecture. The back-end portion of Figure 1.1 shows the resources used to support the cloud and those available to the client for use. Amazon Web Services (AWS) and Google Cloud are examples of companies that provide this type of back-end infrastructure to clients. These providers are capable of providing resources like servers, databases, software, and more to end users on the front end. Instead of a small business having to manage a rack of servers to host its point-of-sale system, that business can pay a cloud provider like Amazon Web Services to do it for them. Clients can pick and choose the tools and resources that they need and only pay for those resources for as long as they need them. This enables end users to reduce computing resource costs and scale their resource usage as needed.

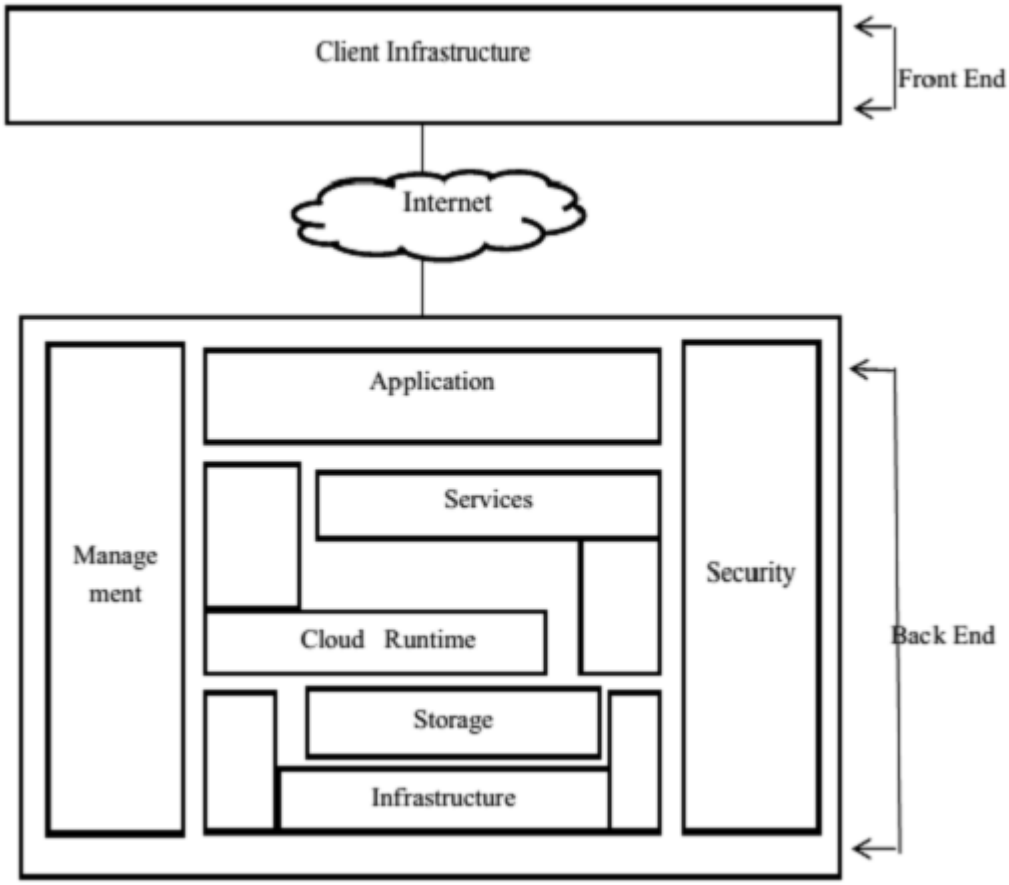



**Figure 1.1:** Cloud Computing Architecture [8].

## 1.3 Containers

There are different ways that cloud-based services can be structured, but one of the most common is through containerized workloads. Containers are a "standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another [9]." These dependencies include everything from the source code and

its associated libraries, to the required runtime environment and system utilities. All of these elements are then compiled into container images, which can be run within container engine tools like Podman or Docker. Figure 1.2 provides a basic diagram of container architecture. One of the primary benefits of containers is

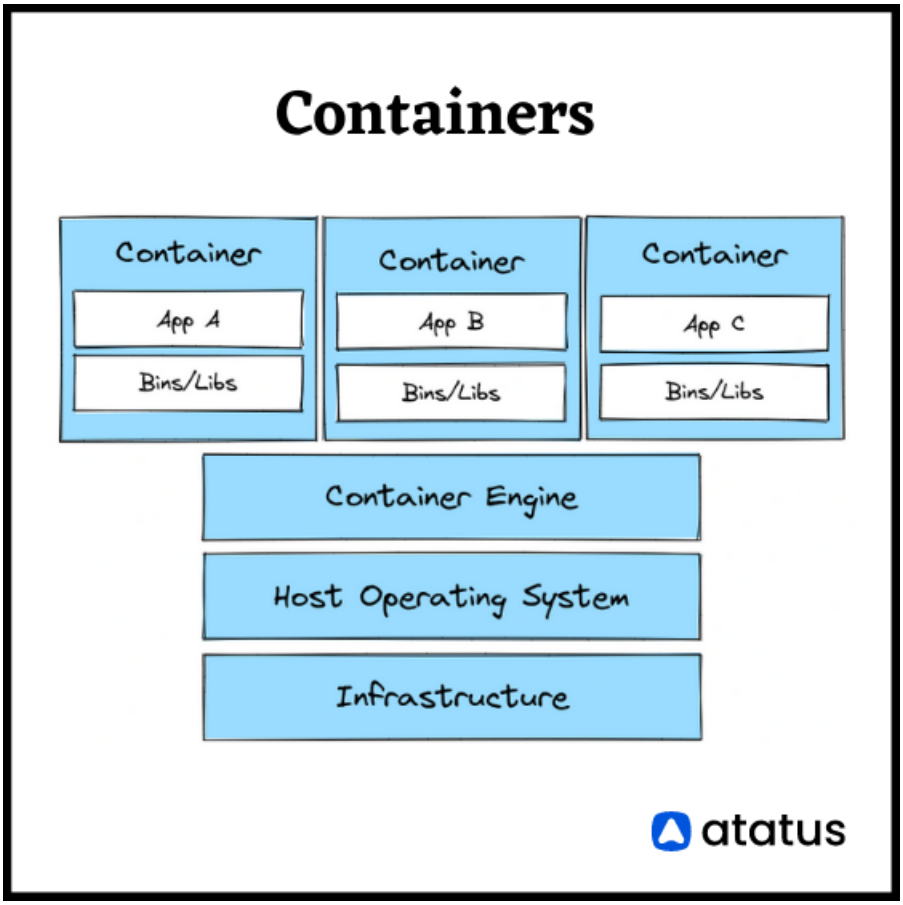



**Figure 1.2:** Container Architecture [10].

that they remove many of the compatibility requirements between applications and underlying operating systems or hardware. As long as the container engine is compatible with the operating system, then there should be no issues running any

container images within the engine. Due to this, containers can run consistently when deployed on Linux, MacOS, or Windows operating systems without having to alter the container image in any way. Containers are also lightweight, so they can be deployed, replicated, or removed from a system with relative ease. This also allows containers to be scaled easily and reliably. Multiple instances of a container image can be created within a single system and, because they are all based on the same underlying image, they will all behave identically.

## 1.4 Container Orchestration

Within the context of cloud-based computing, containers are rarely standalone entities. Individual containers are almost always part of a larger framework that works together to support a specific application or service. For example, an e-commerce website that an end user sees as a single, seamless entity may be comprised of different containerized applications each providing a specific service. One or more containers may be providing payment processing support, hosting the website interface, or storing user account information in a database. In addition to working together to support the end-user service, the deployed containers may need to be scaled up or down based on site traffic, have their software contents updated for security or maintenance purposes, or have functionality added through the deployment of new containers. These realities can make the management of container deployments a challenge, but there are tools designed for container orchestration to help mitigate these challenges. Container orchestration is the process by which many of the tasks that are necessary for implementing con-

tainerized workloads can be automated. Tasks such as deployment, provisioning, and scaling can be handled by these tools.

## 1.5 Kubernetes

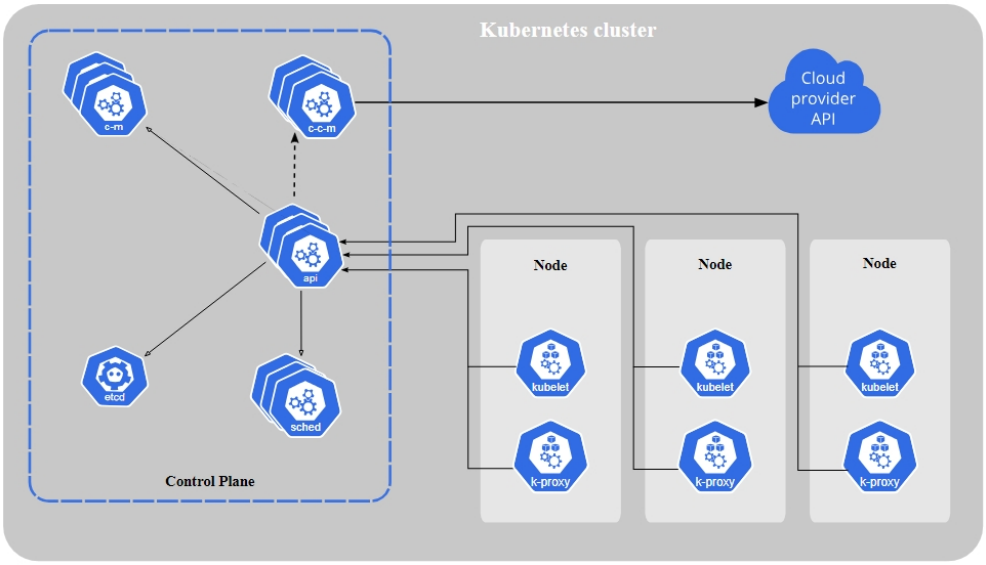
There are many different container orchestration platforms available today, but the most popular is Kubernetes. Kubernetes, or k8s, was originally developed by Google and was released by them as an open-source project in 2014 [11]. Since that time it has been managed by the Cloud Native Computing Foundation. Kubernetes operates as a cluster of worker nodes all managed by a control plane. Figure 1.3 shows a diagram of the basic Kubernetes cluster architecture. Kubernetes clusters use a primary/secondary model where a control plane node manages one or more worker nodes. The control plane, as its name implies, directs the functionality of the entire cluster. Components of the control plane include the *scheduler*, which is responsible for managing resource use across the nodes, *etcd*, a key-value database that stores the configuration data and current state of the cluster, a *control manager* which helps drive the cluster toward the desired state, and the *API server* which facilitates internal and external interfacing with the cluster. Worker nodes are responsible for running the containerized applications deployed across the server. The location of application deployment is determined by the control plane *scheduler* based on the cluster's current resource use, so different nodes may host different applications. The control plane communicates with the worker nodes through *kube-proxy* and *kubelet*. The *kube-proxy* component consists of a proxy server and load balancer and is responsible

for routing traffic and balancing the workload across the node. *Kubelet* is tasked with maintaining the running state of the worker node as directed by the control plane.

Containerized applications are deployed in units called pods, which are distributed across the worker nodes in the cluster as determined by the *scheduler*. A workload can consist of one or more pods depending on the specific application. These deployments can be dynamically scheduled, scaled, updated, and more by the cluster's control plane based on the workload needs at the time. Kubernetes is extremely flexible as the cluster nodes can be deployed on small platform IoT-type devices, virtual machines, physical servers, or cloud services. Kubernetes is not the only cloud orchestration tool available, but it is by far the most widely used.

Cloud-based computing always requires careful observation and management of the resources used to host the cloud platform and its services. This is especially true of devices that are part of the IoT computing framework as they typically have fewer compute resources available than the larger devices in a data center, for example. With minicomputers the power consumption and CPU usage are particularly important, as they greatly impact the performance and longevity of the hardware itself. The actual performance of the applications being hosted in the cloud cluster is also important. If an application or tool hosted within an IoT computing framework has a high latency or slow compute time, then that defeats the purpose of hosting it within that context.

This thesis seeks to characterize the performance and resource consumption of a web server deployed in a lightweight Kubernetes cluster hosted on a series of single-board computers. In Chapter 2 work related to this topic will be presented and various applications and tools used for the research will be discussed. Chapter 3 details the setup of the cluster and the deployment of applications within it. Chapters 4 and 5 detail the experimental methods and results while concluding remarks and comments on future work are presented in Chapter 6.




**(a)** Architecture


**(b)** Legend

**Figure 1.3:** Kubernetes Architecture Diagram [12].

# Chapter 2. Related Works & Background

This chapter presents an overview of research related to the topics studied here and details the tools that were used to perform the experiments. The related works focus specifically on lightweight Kubernetes distributions as well as the performance of web servers hosted on Raspberry Pi devices. The remaining portion of the chapter details the technologies used in the research such as the Raspberry Pi, k3s, and Ansible.

## 2.1 Related Works

Container orchestration tools are a relatively new development, so the amount of research available, particularly in an IoT computing environment, is somewhat limited. Much of the existing research is focused on characterizing different Kubernetes distributions and comparing their performance against one another. Fogli *et al.* [13] compared data transfer and receive rates for vanilla Kubernetes and two lightweight distributions, k3s and KubeEdge, across a variety of tactical network settings. The data showed that KubeEdge performed the best when network conditions were ideal and all three distributions failed under poor conditions. Böhm *et al.* [14] meanwhile measured the average resource utilization of Kubernetes, k3s, and MicroK8s against one another using a cluster

life cycle simulation. They determined that a lightweight Kubernetes distribution can yield better performance than original k8s in certain situations. Specifically, they stated that in fog, edge, and IoT computing scenarios lightweight Kubernetes can be more advantageous than k8s. However, they also recognize that k8s does not consume a significantly larger amount of resources when compared to its lightweight counterparts. A. Kivimäki [15] conducted similar research to Böhm *et al.* and also determined that this was the case. Lastly, Yström *et al.* [16] conducted a study where stress tests were performed on a Minikube cluster and an x-86 embedded system. The results of the study determined that the Minikube cluster had faster local memory access and lower inter-process communication latency under load when compared to the embedded system. Network latency was also tested but the results of the tests were considered to be incomparable to one another. Yström *et al.* did stress that these results only applied to this particular study and could not be generalized to all embedded system and cloud computing situations.

There is also existing research where metrics of interest relevant to this research have been conducted on the Raspberry Pi. Khanjan *et al.* [17] performed research on a Raspberry Pi-hosted web server using Apache Benchmark, the same benchmarking tool used for this thesis. The primary difference was that the web server was hosted on the Raspberry Pi itself rather than in a Kubernetes cluster. The specifics of the web server beyond this were not specified in their research. They compared the performance of the Raspberry Pi web server to MacBooks with Intel and Apple M1 processors, respectively. They found that the CPU

usage of the Raspberry Pi peaked at around 37% when the web server had 10,000 total requests made to it, 10 requests at a time. Additionally, the web server could serve approximately 500 requests per second for the same number of requests and current users. Overall it outperformed the Intel MacBook in terms of CPU usage and the M1 MacBook in terms of requests per second.

With regards to the power consumption of the Raspberry Pi, both Bekaroo and Santokhee [18] and He *et al.* [19] conducted research in this area. Both of these research initiatives used a Raspberry Pi 2 Model B instead of the Raspberry Pi 4 Model B. However, the devices are analogous to one another with the Raspberry Pi 4 being the newer version of the Model B. In their research, Bekaroo and Santokhee measured the power consumption of the Raspberry Pi while it performed various common operations such as booting up, downloading a file, streaming a video, word processing, etc. They found that tasks requiring internet connectivity required more power than those that could be performed offline. Overall, the power consumption of the device averaged 3.68 Watts for internet-connected tasks, and 2.58 Watts for offline tasks. These power consumption results were less than that of a laptop and desktop computer and comparable to those of a Samsung Galaxy S6 Edge smartphone. He *et al.* [19] focused on measuring the power consumption when running several graphics processing-related benchmarks. The benchmarks run by this group determined that the Raspberry Pi consumed between 1.72 Watts while idle, to 1.94 Watts when running a 3D modeling benchmark. They concluded that power consumption increased as the

frame rendering rate increased, but that the overall energy used per frame could be lower due to less total time spent processing the graphics.

## 2.2 Experimental Platform

### 2.2.1 k3s: A Lightweight Kubernetes Distribution

The original Kubernetes project, while extremely robust, is not ideal for the IoT computing environment. This version of Kubernetes was designed for implementing large clusters comprised of many nodes in an environment that necessitates high availability, scaling, and performance. It has quite a large number of dependencies and often requires multiple master and auxiliary nodes to take full advantage of its features. Furthermore, many of these features are designed for large-scale deployments that support machine learning or high-performance computing and would be ill-suited to the IoT environment. When compared to the average server on which k8s would typically be deployed, a device like the Raspberry Pi 4 has much less capacity in terms of memory, computational power, and robustness. Fortunately, other lightweight versions of Kubernetes have been developed that are much better suited to small-footprint devices like the Raspberry Pi. One of the most popular lightweight Kubernetes projects is k3s [20]. Developed by Rancher Labs, k3s was designed to be used in IoT-like settings, and can easily be deployed on single-board computers. The software is packaged as a single binary file that is less than 100 MB in size, but is still a fully compliant version of Kubernetes certified by the Cloud Native Computing Foundation [21].

The primary difference between k8s and k3s is that k3s has been designed to have a much smaller memory footprint than the original. For example, the control plane in standard Kubernetes is generally distributed across a series of devices, each running a different process that is part of the control plane. In contrast, k3s runs all of the control plane elements within a single process on a single server node [22]. Figure 2.1 shows the architecture of a server and agent node. The left-hand side of the figure shows the server node and the right-

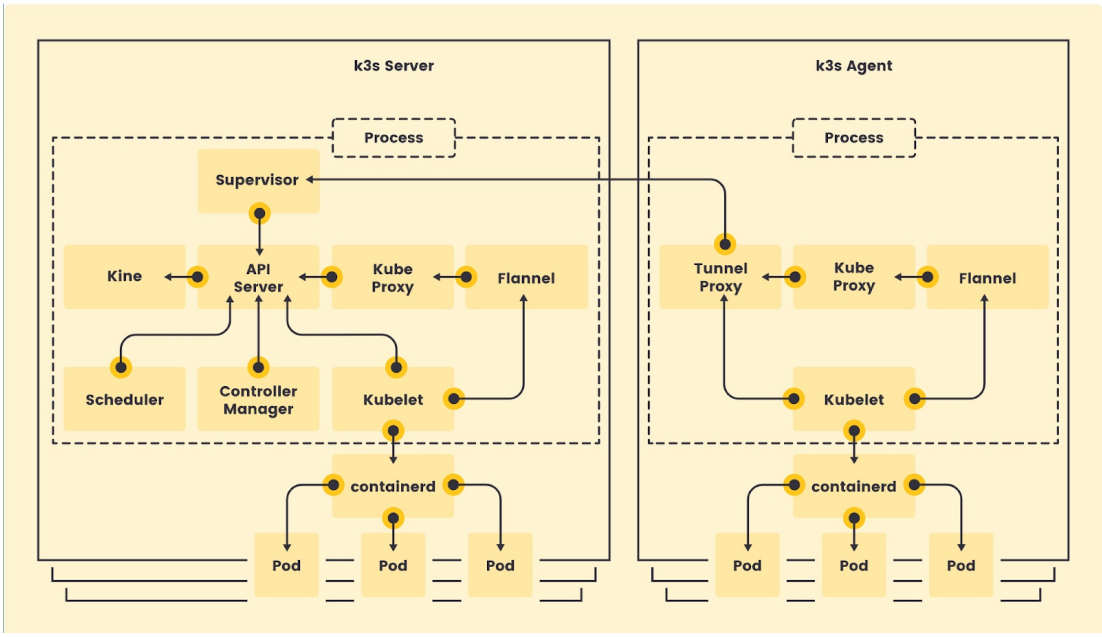



**Figure 2.1:** k3s Architecture: Server and Agent Node Diagram [23].

hand side shows the agent node. On the server node all of the elements that comprise the control plane are running within the server node process. In vanilla Kubernetes these control plane elements would be distributed across multiple

different control nodes. Furthermore, the server node in k3s is also capable of running *containerd*, the standard container runtime environment through which pods are deployed and managed. The presence of *containerd* on the server node means that it also has the same capabilities as the agent nodes, meaning that end-user applications can be deployed on the server node as well. This contrasts with vanilla Kubernetes in which the control plane nodes are completely separate from the worker nodes and cannot run end-user applications. This feature means that k3s can be deployed on an individual single-board computer rather than requiring multiple devices for baseline deployment, making it ideal for IoT settings. K3s also has other features that reduce its memory footprint such as its use of SQLite as the default back-end database for the control plane. If this is not suitable for the developer, however, a different database such as MariaDB, MySQL, or *etcd* can be substituted for SQLite. Kubernetes on the other hand, only supports *etcd* for its database and a different database application cannot be used [22]. This flexibility not only applies to the back-end database but to other elements of the k3s service as well. An external Docker installation can replace the *containerd* container runtime for example and all of the elements of the control plane such as the default DNS server or ingress controller can be switched off and equivalent applications installed in their place [22]. These features make k3s highly flexible for such a small package. All told it only requires a single-core CPU with 512 MB of RAM to run at a minimum. The preferred computer specifications are only a two-core CPU with 1 GB of RAM [23].

The k3s cluster deployed for this research was deployed as a single-server node with two associated worker nodes. The default control plane elements and *containerd* runtime environment were used. Figure 2.2 shows an example of a server/agent node configuration in k3s. As Figure 2.3 shows all management of

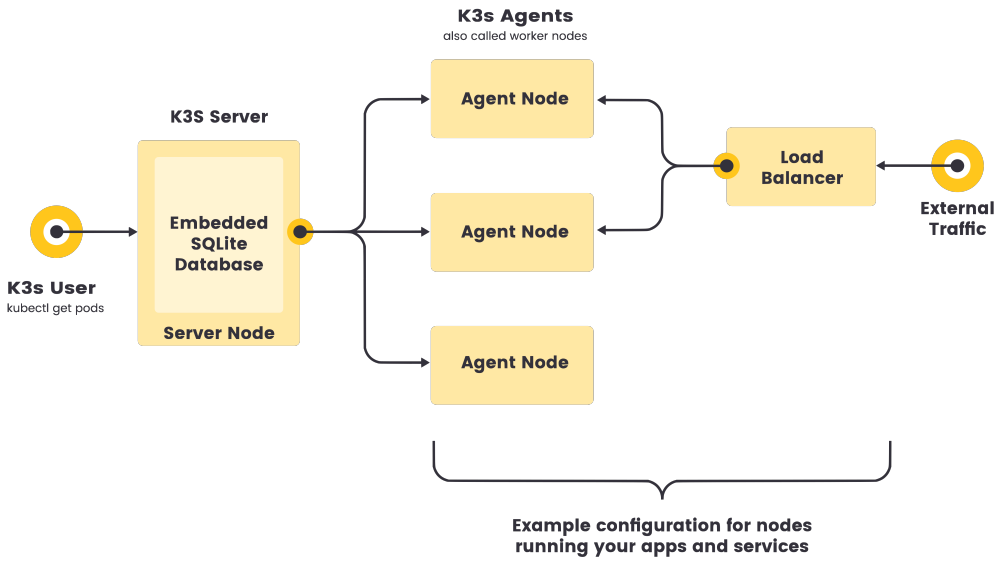



**Figure 2.2:** k3s Architecture: Server with Three Agent Nodes [24].

the cluster in k3s is handled through the server node. Communication with the control plane is done through a command line tool called kubectl which is the primary way that the cluster is communicated with. For a single server node there can be multiple agent nodes that run the end-user services and applications. The external traffic from end-users using the hosted applications runs directly to those applications on the agent nodes. If the application in question is hosted across multiple nodes the traffic may be routed through a load balancer to distribute

the work evenly, though this is typically only necessary in situations where high cluster availability is critical.

### 2.2.2 Raspberry Pi

There are many different types of microcomputers available for purchase today, but one of the most ubiquitous is the Raspberry Pi. A series of small, single-board computers, the Raspberry Pi was developed in the United Kingdom by the Raspberry Pi Foundation. In 2012 the first commercially available Raspberry Pi, the Model B, was released to the public. Since that time new versions of the Raspberry Pi with different models have been released, most recently the Raspberry Pi 5 in October 2023. This research was conducted using the Raspberry Pi 4 Model B, originally released in June of 2019 [25]. The Raspberry Pi 4 is available with 2, 4, or 8 GB of RAM and comes equipped with a 1.8 GHz quad-core 64-bit ARM processor. The Pi 4 has multiple USB ports and two micro HDMI ports that can be used to connect a computer mouse, keyboard, and monitor directly to the device. The operating system can be loaded to the computer through one of the USB ports or the included Micro-SD card slot [26]. Figure 2.1 shows an image of a Raspberry Pi 4 Model B. The Figure also shows a 1GB RAM model of the Raspberry Pi 4, but this model was discontinued shortly after the release of the Pi 4. The 2, 4, and 8 GB models are still manufactured [25]. The entire series of Raspberry Pi computers were designed with cost in mind, and the Pi 4 typically costs between 35 and 75 USD depending on the model that is purchased [26]. This low cost has made the Raspberry Pi one of the most popular

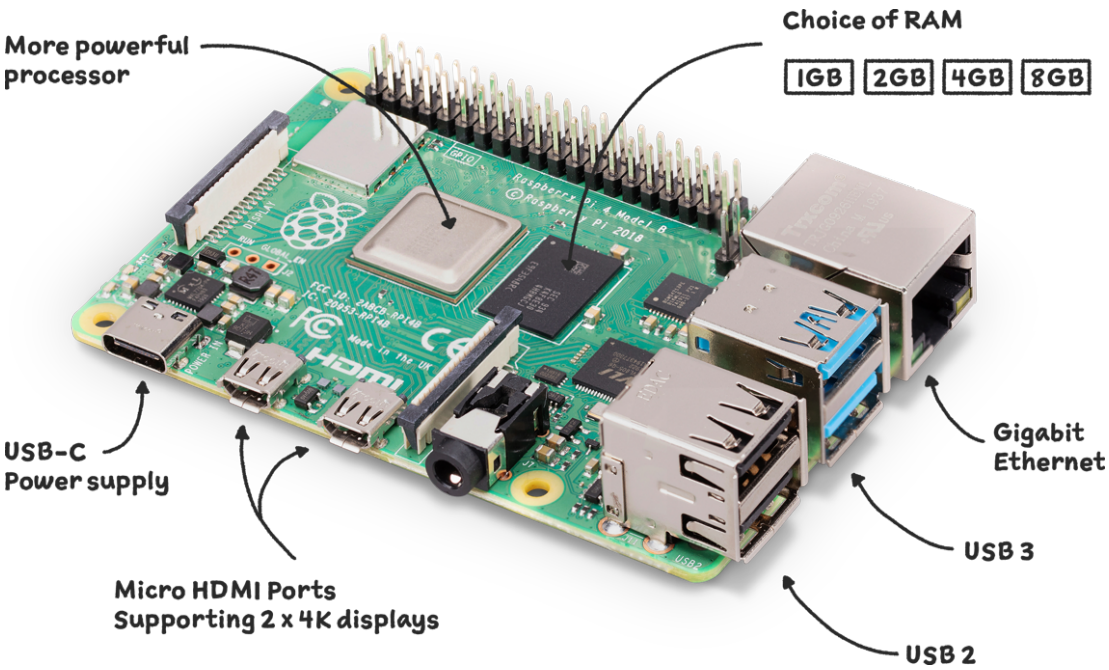



**Figure 2.3:** Raspberry Pi 4 Model B[26].

minicomputers on the market today, with it being in everything from industrial and defense applications to hobbyist home labs. Depending on the Kubernetes distribution, one or more Raspberry Pi devices can also be a good environment for hosting an IoT computing cluster. Many hobbyists who host local small platform clusters for use within their homes utilize the Raspberry Pi as the underlying hardware for their clusters. As a quad-core computer with 8 GB of RAM, the Raspberry Pi was more than capable of hosting k3s using the preferred computer specifications provided by Rancher, hence its use in this research.

## 2.3 Ansible Automation Tool

While k3s is much simpler than the original version of Kubernetes, that does not mean that working with it is always easy. The installation process can still be quite complex and time-consuming, as the server node and all associated worker nodes have to be independently configured and added to the cluster. In the case of a small deployment, this may not be an issue, but as the individual nodes in a cluster grow, the manual configuration process can become prohibitive. This is where tools like Ansible come in handy. Ansible is tool that is used to automate various IT-related jobs like provisioning, application deployment, and configuration management. Figure 2.3 shows an example of the workflow of Ansible and examples of the tasks that it can be used to automate. As Figure 2.3 shows, the software works via the use of Ansible playbooks, which are scripts that contain the instructions necessary to complete a specific task or set of tasks. Via Ansible, these playbooks are capable of automating a variety of tasks from the provisioning of virtual machines to the configuration of network switches [27]. Automation tools like this are particularly valuable for back-end developers who do cluster management, as they allow for streamlining of tasks that can be large time sinks if they have to be done manually. K3s provides an Ansible playbook on their GitHub page for the deployment of a cluster [28]. Once the playbook has been run the tool will then return a summary to the user of every task that was done and whether it was completed successfully. For a small cluster as was used in this research, the entire process only takes one to two minutes from start to

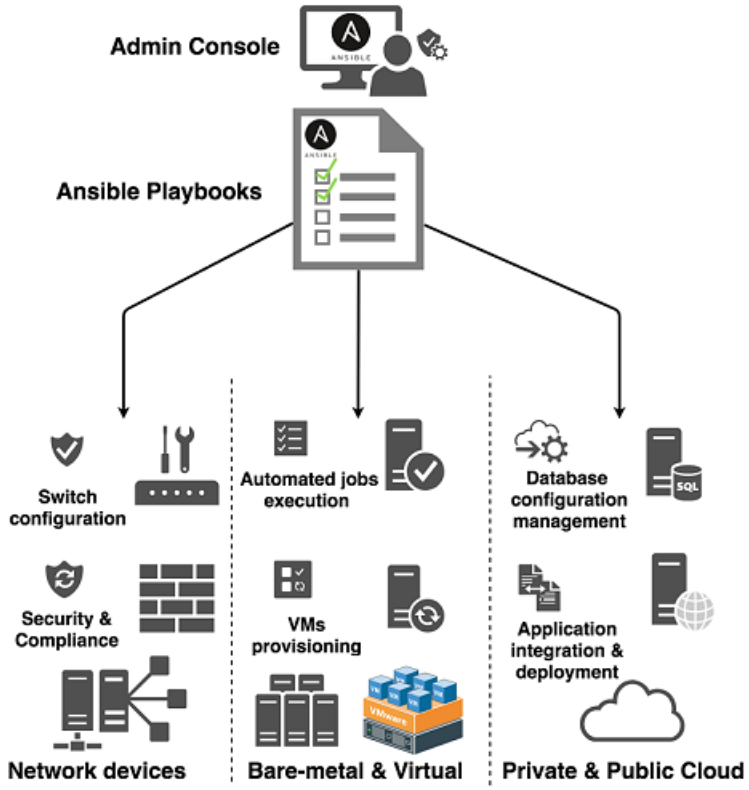



**Figure 2.4:** Ansible Automation Tool [27].

finish. If done manually the process would require much more time as each device would have to be configured, deployed, attached to the cluster, and checked for configuration correctness manually. Appendix B provides detailed instructions for installing Ansible and deploying a k3s cluster using the k3s Ansible playbook.

## 2.4 Resource Monitoring

### 2.4.1 Prometheus

One of the primary downsides of any distribution of Kubernetes is that there is no default way to monitor the performance and health of a cluster. While the cluster is running on physical hardware such as servers or minicomputers, the cluster itself is abstracted away from the underlying hardware, so the performance of one is not directly analogous to the other. Thus monitoring the underlying hardware does not necessarily provide much information for what is going on within the cluster. Depending on what the cluster is doing, the administrator managing it may want detailed information about the status of the nodes or the pods they are hosting. Although Kubernetes does not offer an automatic way to monitor these metrics, it is capable of providing them to the administrator when asked. This is where Prometheus, an open-source application that is used for system monitoring and alerting, enters the picture. Prometheus works by collecting metrics from its targets via HTTP requests. Once the target responds with the required data, it is parsed and then stored in a time series database, which can then be accessed by administrators for monitoring purposes [29]. Thresholds can be set for specific metrics being monitored, and the system can notify administrators through alert management tools if these thresholds are exceeded. While Prometheus can monitor more than just Kubernetes clusters, it is often the default choice used by administrators and engineers working with container orchestration tools. After Kubernetes, Prometheus was the second project in-

cubated by the CNFC, so, much like k3s, it is considered reliable in the cloud computing space [29].

### 2.4.2 Grafana

The major downside of Prometheus is that it does not have a good way of displaying the data that it collects. Data are viewable within Prometheus, but only as lists in the time series database. There are no graphs, trend lines, etc. that the tool provides. Due to this, Prometheus is often deployed alongside Grafana, which is an open-source analytics and visualization tool [30]. The data collected by Prometheus are sent to the Grafana application which can then display the data in graphical formats in dashboards. Dashboards are customizable so users can configure them as they see fit and multiple dashboards can be set up for different subsets of data. Figure 2.5 shows an example Grafana dashboard. Grafana is also hosted as a web application, so users can view dashboards via any web browser, making it convenient to access and use.

Together Prometheus and Grafana provide a robust method for monitoring Kubernetes metrics in real time. Typically both applications are deployed within Kubernetes itself, running as a collection of pods on the cluster. Once deployed, the Grafana dashboard can then be accessed via a web browser by using the ingress address provided by Kubernetes. Figure 2.6 shows a diagram of how Kubernetes, Prometheus, and Grafana all work together within the context of a cluster. In Figure 2.6 the monitoring tools are being hosted within an original


**Figure 2.5:** Grafana Dashboard Example [31].

Kubernetes cluster but the deployment of Prometheus and Grafana within a k3s context is identical to the diagram as well.

### 2.4.3 Waveshare Power Monitor HAT

The final tool used in montoring the experiments was the Waveshare Power Monitor HAT, shown in Figure 2.7 [33].

This peripheral device is compatible with the Raspberry Pi and is designed to fit over the GPIO header on the device's backplane. It has 4 independent input channels that are used to measure the current or power of any connected device. The measurements are then transmitted to the Raspberry Pi via the I2C/SMBus. The module uses a 0.1Ω sampling resistor and a 12-bit ADC to obtain current and voltage measurements and is capable of calculating power directly via onboard

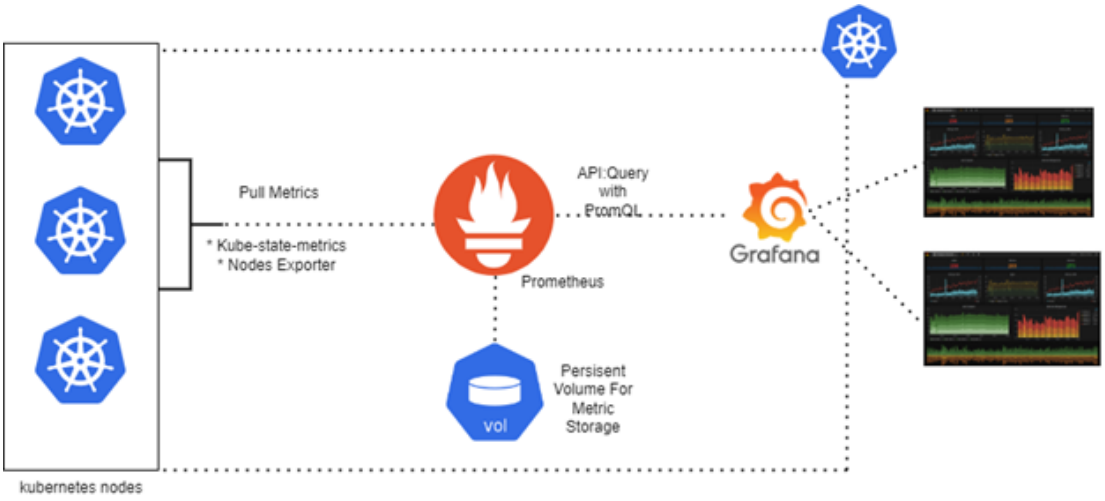



**Figure 2.6:** Cluster Monitoring Diagram [32].

multiply registers. It is capable of measuring up to 3.2A and has a voltage range of 0 to 26V.

## 2.5 Load Testing

In characterizing the cluster it was necessary to induce load on it somehow. First, a target was needed that could be hosted on the server and be the subject of the created load. A web server was chosen as the target application for the research tests. This was a suitable target as a web server is an object that would typically be hosted in a cloud-based cluster and connection requests could be made to the server to simulate web traffic. There are several different platforms that are available for hosting a web server, and Drupal was selected as the platform of choice.

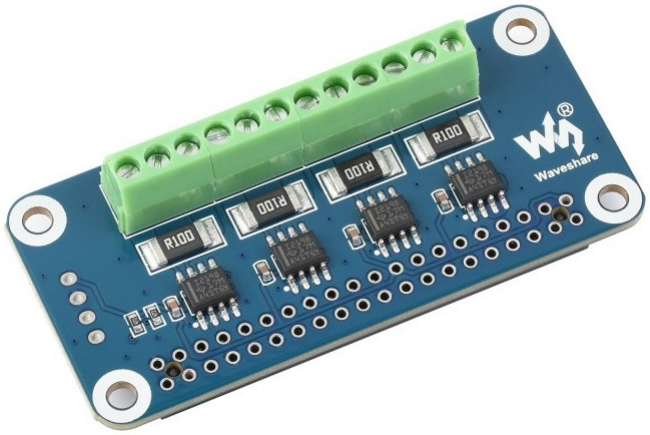



**Figure 2.7:** Waveshare Power Monitor HAT [33].

### 2.5.1 Drupal Web Server

Drupal was chosen as the platform for hosting the web server while MariaDB was selected as the database [34][35]. Drupal is a free and open-source Content Management System (CMS) most commonly used for hosting websites. Drupal was initially released in 2001 and has been actively maintained since that time, so it is considered to be a well-regarded CMS tool. It is written in PHP and at 140 MB in size it is also small enough to be easily deployed in a cluster environment hosted on a minicomputer like the Raspberry Pi. MariaDB was selected as the database server as it was recommended in the official Drupal documentation.

### 2.5.2 Apache Benchmark

The tool that was used for creating a load on the Drupal web server was ab, which stands for Apache Benchmark. Apache Benchmark is a free-to-use tool developed by Apache that can be used for benchmarking HTTP servers. As of

May 22 2024, OpenBenchmarking.org reports that Apache Benchmark has been installed over 200,000 times and used to conduct over 600,000 tests [36], so it is considered reliable for this purpose. The Drupal CMS used in this research utilized Apache in its web server, so the use of this tool was appropriate. The tool is run from the command-line interface and is simple to use. All a user has to do is specify the total number of requests they want to make to a server, the number of concurrent requests they wish to perform, and the IP address of the server they wish to benchmark. The tool will then send the requests to the target server and keep track of the total number of requests served, the number of requests served per second, etc. and return this information to the user after the test. Figure 2.8 shows the results of an example benchmarking test where the target is google.com.

The command that was run to produce these results was ".\ab -n 25 -c 1 -r http://google.com." This command tells the tool to perform a total of 25 requests to google.com with concurrency level 1, meaning that only one request will be made at a time. The -r option is used to keep the tool from exiting on a socket receive error [37]. After completing the benchmark test, the tool outputs the results collected during the test. These results include the total time taken to complete the test, the number of completed and failed requests, average requests per second and time per request, as well as the percentage of the requests severed within a certain time. All of these metrics provide insight to the user on the performance of the server, particularly the number of requests per second the server is capable of serving. Running these tests also places load onto the server

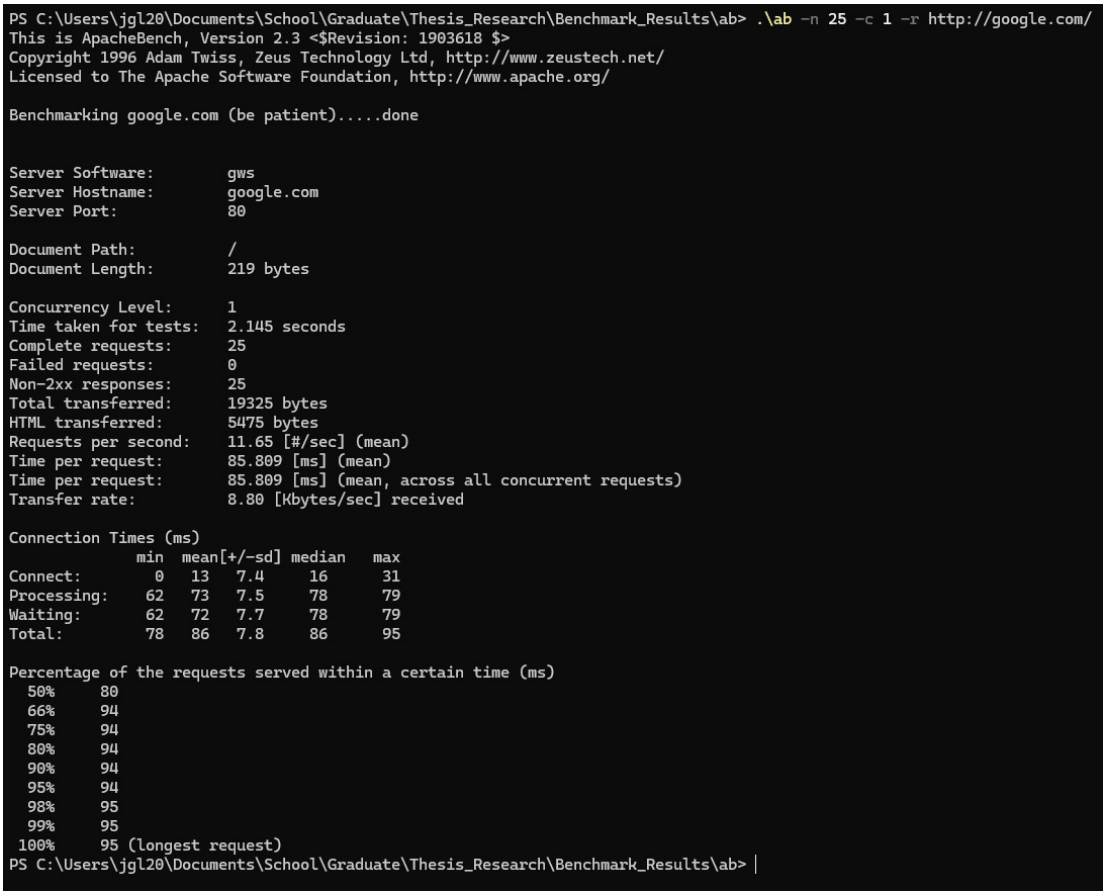



**Figure 2.8:** Apache Benchmark Example Output.

which, in turn, should increase the CPU usage and power consumption of the nodes in the cluster. Apache Benchmark does not return these metrics, but they were measured during the test time using methods external to the benchmark tool. Apache Benchmark was run from a machine on the same network as the k3s cluster but outside of the cluster itself. Thus the running of the tool placed no additional load onto the cluster during the tests.

# Chapter 3. Experiment Setup & Components

The overall structure of the experimental setup can be seen in Figure 3.1. The details of how each of these components were set up are detailed in the subsequent sections.

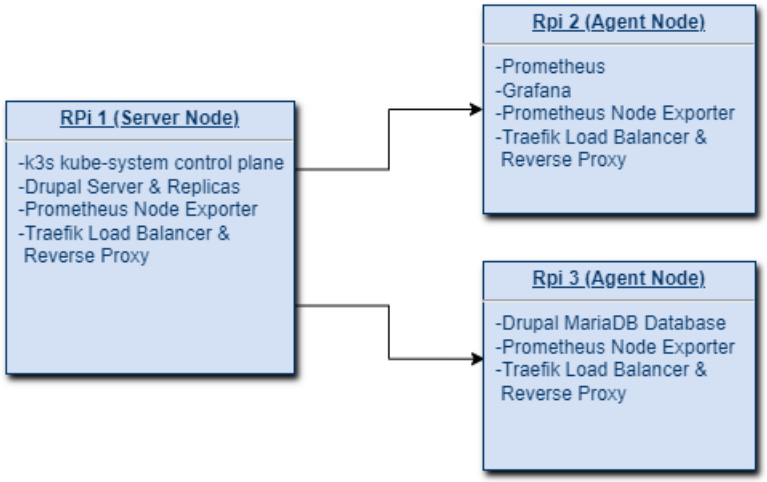



**Figure 3.1:** Services Running on Each Cluster Node.

## 3.1 Raspberry Pi Hardware Configuration

The Raspberry Pi computers used in this research were all the Raspberry Pi Model B with 8 GB of RAM and were all purchased from CanaKit. The operating system used for each device was the 64-bit version of Raspberry Pi OS which was booted from a MicroSD card in the Raspberry Pi board. The bootable

version of the operating system was placed onto the microSD cards using the Raspberry Pi Imager tool. Once the boot process was completed the hostname for each device was configured and the SSH server enabled. For the initial setup of the devices a mouse, keyboard, and monitor were required. These were connected via the Rasberry Pi's onboard USB and micro HDMI ports. Once the SSH servers were set up an SSH connection via the Linux Bash shell was the primary way in which all of the devices were accessed. Figure 3.2 shows one of the Raspberry Pi devices that was used.

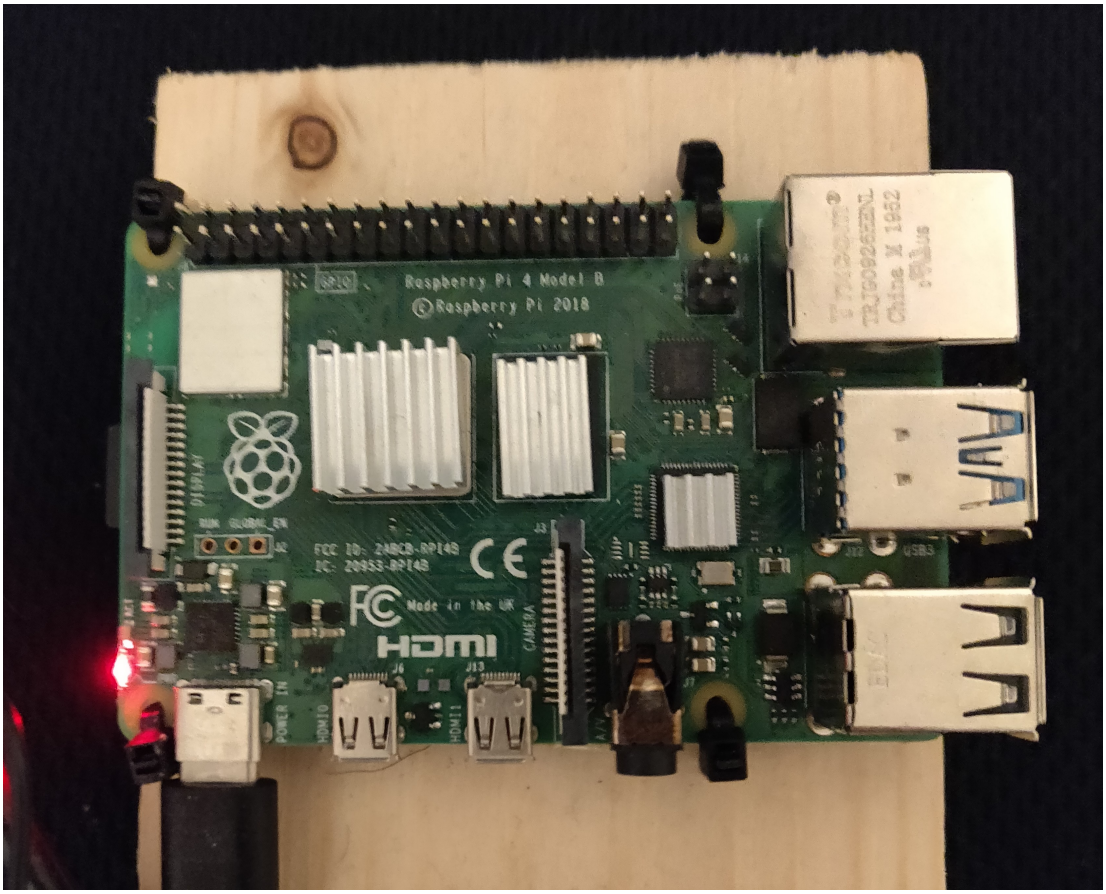



**Figure 3.2:** Raspberry Pi Model B Used in Research.

In total, 3 Raspberry Pi devices were used for this research, with one functioning as the server node and the other two working as agent nodes. The server node was designated as Node 1 while the agent nodes were designated as Nodes 2 through 3. Figure 3.3 shows a picture of all four devices that comprise the cluster.

**Figure 3.3:** Raspberry Pi Cluster.

Additional details regarding the specific versions of Raspberry Pi OS and Raspberry Pi Imager used as well as step-by-step instructions for installation can be found in Appendix A.

## 3.2 K3s Cluster Configuration & Object Creation

### 3.2.1 Cluster Deployment Using Ansible

Once all of the Raspberry Pi devices were booted and configured, installation of the k3s cluster began. To install the k3s environment across the cluster and make sure it was configured properly, the software automation tool Ansible was used. Rancher, the original makers of k3s, have an Ansible playbook on their GitHub that can be used to create and provision an N-node cluster [28]. To use the playbook, all a user has to do is specify the IP addresses of the server and agent nodes and ensure that passwordless SSH access and root privileges have been set up for all devices hosting a node. Once these steps are completed, the user runs the playbook, and all steps of setting up the cluster, from downloading the k3s binary to configuring each node, is done automatically. Since the k3s cluster is lightweight this process only takes a few minutes. At the end of the process, Ansible returns a summary of all tasks performed and indicates whether or not they were completed successfully. The GitHub repository also provides playbooks for safely shutting down and decommissioning the cluster as well. Appendix B details the steps for deploying the cluster.

### 3.2.2 Kubernetes Objects & Manifests

With the k3s cluster configured and deployed, Kubernetes objects can be deployed within the cluster framework. A Kubernetes object describes entities deployed within a Kubernetes cluster. An object details what container applica-

tions are running on a node, the policies and permissions of those applications, and the resources available to them. These various characteristics of an object are described in detail in an object specification that is part of a file known as a manifest. A manifest is a JSON or YAML formatted file that describes the Kubernetes object in its desired state and an object is created by applying the manifest file to the cluster. An example of a simple manifest file can be seen in Figure 3.4.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

**Figure 3.4:** Kubernetes Manifest Example [38].

Figure 3.4 shows an example YAML manifest file for an Nginx web server that can be run in Kubernetes. The file details that the object will be a Deploy-

ment named nginx-deployment and that it will contain 2 identical pods based on the manifest template. The template, in turn, details that the containers within the pods will run an Nginx image pulled from the repository nginx with tag 1.14.2. The default repository for Kubernetes is the Docker public registry. If Kubernetes can pull the image from this location successfully, it will then build the deployment using it and the other specifications within the manifest file. All Kubernetes objects are created based on manifest files like this, though many are more complex than the example in the figure.

## 3.3 Deploying Applications on the Cluster

### 3.3.1 Prometheus & Grafana

Prometheus and Grafana were the means through which the data were collected from the cluster. As previously described, Prometheus is the tool that gathers and aggregates data from the cluster and Grafana displays that data in a user-friendly manner. These are the two tools that are most commonly used for monitoring of a Kubernetes cluster. The monitoring stack that was used for this research was created by Carlos Eduardo [39] and the repository was hosted on GitHub [40]. This repository was selected as it was highly starred on GitHub and was compatible with ARM devices and the k3s framework. Launching the monitoring stack was accomplished by editing the project's vars.jsonnet file to configure parameters for k3s, building the project using Make, and then applying the created manifests to the cluster. Once the cluster was

up and running, the Grafana dashboard was accessed by typing the ingress IP address "grafana.192.168.1.198.nip.io" into any web browser that was connected to the same network as the cluster. Figure 3.5 shows an image of the Grafana dashboard used for this research. All pods running the Prometheus and Grafana applications were on agent Node 2 of the cluster.


**Figure 3.5:** Cluster Grafana Dashboard.

The dashboard displayed real-time data collected by Prometheus, and data could be downloaded from the dashboard as comma-separated values (CSV) files. The primary metric collected by the tool used for this research was the CPU usage of each node in the cluster.

### 3.3.2 Drupal Server

Like the Grafana and Prometheus objects, the web server object was applied to the cluster using a manifest. The pod running the Drupal front end was allocated to Node 1 by the cluster scheduler while the pod running MariaDB was

allocated to Node 3. Like the Grafana dashboard, the web server was accessible from any web browser on the same network as the server at the IP address "drupal.192.168.1.197.nip.io." The landing page of the Drupal site is shown in Figure 3.6.

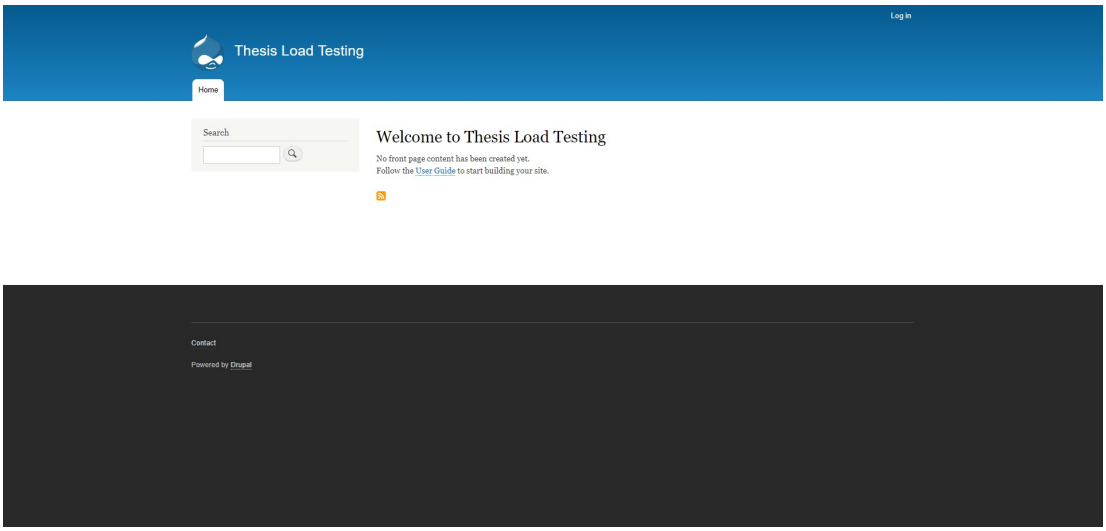



**Figure 3.6:** Drupal Site Landing Page.

Other than naming the page "Thesis Load Testing" no additional features were added to the web server as its existence, not capabilities, were all that was necessary for research tasks.

Initially, only one pod supporting the Drupal web page was deployed to the server. Kubernetes Deployments support a feature called ReplicaSets, where a set of identical replica pods are running on the server at any given time. The number of replicas can be specified whenever the Deployment is created, or it can be manually scaled up or down during runtime. This action can also be done automatically using the Horizontal Pod Autoscaler (HPA), where more replicas

are brought online if the current pod(s) load exceeds a specified threshold. Unfortunately, the HPA could not be used in this research due to compatibility issues with the vanilla instance of k3s and the Raspberry Pi. Thus, the pods were scaled up manually when running experiments for this research. Between one and seven total replicas of the Drupal pod were used in the trials conducted for this research. Seven was the maximum number of replicas used as additional pods above this threshold were unstable when brought online.

### 3.4 Power Monitor Setup

For the Raspberry Pi to be able to measure its power consumption, the power cable was spliced and connected to input channel 4 of the monitor module. This was done for all of the devices in the cluster. To take the power measurements a Python script provided by the vendor was modified so that the measurements along with timestamps for each would be written out to a CSV file. After a measurement was written to the output file, the next measurement would be written after a delay of 0.1 seconds, resulting in 10 measurements per second. This method of obtaining the measurements was used for each Raspberry Pi board in the cluster. Appendix C contains the Python source code used for this task.

# Chapter 4. Experimental Methods & Metrics

The experimental trials conducted for this research focused on inducing a load onto the Drupal web page and measuring the performance of the server, the CPU usage, and the power consumption of each node in the cluster.

## 4.1 Apache Benchmark Metrics

Load was created on the server using the Apache Benchmark tool and the server performance was measured directly by the tool. The primary metric of interest collected from the tool were the average Time per Request (TPR) and Transfer Rate returned by the tool. These metrics were measured in milliseconds (ms) and Kilobytes per second (Kbytes/s), respectively. Additional metrics of interest were the number of completed and failed responses and number of non-2xx responses returned by the server. Apache Benchmark calculates the Time per Request and Transfer Rate using the following equations [37]:

$$Time\ per\ Request = \frac{timetaken * 1000}{done} \tag{4.1}$$

$$Transfer\ Rate = \frac{totalread}{1024 * timetaken}. \tag{4.2}$$

In Equation 4.1 *timetaken* is the total amount of time required to complete all of the connect requests in seconds and *done* is the number of requests completed by the benchmark. For Equation 4.2 *totalread* is the total number of bytes read while *timetaken* is the same as in Equation 4.1.

## 4.2 CPU Usage & Power Metrics

The CPU usage data were obtained from Prometheus and Grafana, and the power measurements were collected using the power modules on each Raspberry Pi. For each trial power readings and CPU usage data were collected for all 3 nodes in the cluster. Power measurements were collected every 0.1 seconds while CPU usage measurements were collected every three seconds. The slower collection of the CPU usage data had to do with limitations in Prometheus's data collection speed and Grafana's ability to log that data. These tools are primarily used for cluster monitoring and alerting, not detailed data collection, so measurements every few seconds are often considered sufficient. Three seconds proved to be the fastest rate at which Grafana could log data without encountering performance issues on the cluster. For each trial, measurement of the CPU usage and power consumption was done both before and after the Apache Benchmark tool was run. This was done to establish a before and after baseline for these metrics

that could easily be distinguished from the measurements that occurred while the cluster was under load. In addition to these before and after measurements, an additional two-minute run was conducted where no load was induced on the cluster by the benchmark tool. This was done to obtain a series of baseline measurements for power consumption and CPU usage to use as a reference for the trial data.

### 4.3 Trial Methods

The duration of each trial was dictated by the total number of requests and the concurrency level specified in the Apache Benchmark terminal command. Holding the concurrency level constant and increasing the numbers of requests would lengthen a trial, while increasing the concurrency level for a fixed number of requests would reduce the length of a trial up to a certain threshold. Beyond that threshold, the trials would take longer due to the load creating so many concurrent requests put on the machine running the benchmark program. The number of requests was set such that each trial would take approximately two minutes to run for each concurrency level. Concurrency levels were increased from 1 by increments of 5 up to 30. In addition to the two-minute trials, an "endurance run" was also performed where, for a concurrency level of 15, the number of requests was set so that the test would last for 15 minutes. Table 4.1 details the concurrency level and number of requests that comprised each trial conducted during the experiments.

**Table 4.1:** Trials with their associated Requests and Concurrency Level.

| Trial | Number of Requests | Concurrency |
|---|---|---|
| Baseline | - | - |
| 1 | 500 | 1 |
| 2 | 1000 | 5 |
| 3 | 1800 | 10 |
| 4 | 1700 | 15 |
| 5 | 1600 | 20 |
| 6 | 1600 | 25 |
| 7 | 1600 | 30 |
| Endurance | 12000 | 15 |

The set of trials detailed in Table 4.1 was conducted for all Drupal pod replica quantities ranging from one to seven. In total data were collected for 63 trials across all the replica sets.

## 4.4 WordPress Server

In the initial stages of the research, a fourth Raspberry Pi was part of the k3s cluster. This device was identical to the other three boards and was hosting a website built using the WordPress content management system. This was initially

included to see if a website hosted using a different CMS would have any impact on the experimental results. The experiments run for the WordPress server were identical to those run for the Drupal server. However, tests on this server produced results that were not meaningfully different from the Drupal server results. It was also unstable compared to the Drupal server, frequently producing error codes in response to requests. As a result, the data produced from these trials were only used to contrast with the data obtained from the Drupal server.

## 4.5 Raspberrry Pi Based Drupal Server

Since the fourth Raspberry Pi was not needed in the k3s cluster, it was instead used as a local host for an additional Drupal server. The rationale for this was that the same tests could be run on this and the k3s-based server and the resulting metrics could be compared to one another. Thus the metrics gathered for the k3s cluster would not exist in a vacuum and could be compared to a more conventional setup.

The Raspberry Pi-based server was set up identically to the k3s-based server, using the same Apache, PHP, MariaDB, and Drupal versions. A landing page identical to that shown in Figure 3.6 was set up and was accessible on the same network as the k3s cluster-based server. The same tests shown in Table 4.1 were performed on this server using the Apache Benchmark tool. The metrics were gathered using Apache Benchmark, the WaveShare Power Monitor and, Prometheus/Grafana. Prometheus and Grafana could be used because the fourth

node still had the monitoring stack installed from when it was part of the original cluster. The same computer was used to run the Apache Benchmark tool as well.

# Chapter 5. Results and Discussion

The results of the experimental trials will now be examined. All tables and figures were generated using LaTeX within the Overleaf LaTeX Editor application.

## 5.1 Verification of Measurement Precision

In conducting experimental trials it is often good to repeat them multiple times with the same parameters and record the results of each attempt. The results should be the same or similar if the experiment setup, parameters, and data collection methods are reliable. If the results of repeated experiments are not similar, this indicates that the experiment does not produce precise results and methods may need to be re-evaluated. For this research, trials one and two as shown in Table 4.1 were repeated three times each for a ReplicaSet of one. This was done to measure the difference between the measurements obtained for each trial. If the measurements were precise this would indicate that the experimental methods were reliable, meaning that subsequent experiments need not be repeated. Conversely, if the results were not precise then the experimental methods would need to be re-evaluated. The results for all measured metrics and the maximum percent difference between the measurements for trial one are shown in Table 5.1

**Table 5.1:** Results from Repeated Trial One, Concurrency 1, ReplicaSet 1.

| **Metric** | **Trial 1** | **Trial 2** | **Trial 3** | **Max. % Diff.** |
|---|---|---|---|---|
| Transfer Rate (Kbytes/s) | 33.18 | 34.28 | 33.99 | 3.26 |
| Av. CPU Usage (%), N1 | 12.66 | 12.13 | 12.53 | 4.28 |
| Av. CPU Usage (%), N2 | 11.42 | 11.03 | 11.59 | 4.95 |
| Av. CPU Usage (%), N3 | 9.28 | 9.35 | 9.21 | 1.51 |
| Av. Power Usage (W), N1 | 3.19 | 3.18 | 3.20 | 0.63 |
| Av. Power Usage (W), N2 | 2.95 | 2.96 | 2.98 | 1.01 |
| Av. Power Usage (W), N3 | 2.96 | 2.97 | 3.01 | 1.68 |

The results in Table 5.1 indicate that the experiments produce precise results, with the largest difference in values measured being 4.95% for the CPU usage of Node 1. This percentage difference corresponded to an actual maximum measured value of 0.56% CPU usage, which was considered to be an acceptable difference between the measurements produced by each trial. As the table shows, all other values measured had a maximum percentage difference of less than 4.95% indicating that they too were within an acceptable difference of one another.

Trial two was also repeated three times to ensure that different trials would produce equally precise results when repeated. Table 5.2 shows the results of this trial.

**Table 5.2:** Results from Repeated Trial Two, Concurrency 5, ReplicaSet 1.

| **Metric** | **Trial 1** | **Trial 2** | **Trial 3** | **Max. % Diff.** |
|---|---|---|---|---|
| Transfer Rate (Kbytes/s) | 72.18 | 70.52 | 71.31 | 2.33 |
| Av. CPU Usage (%), N1 | 19.07 | 18.72 | 18.72 | 1.85 |
| Av. CPU Usage (%), N2 | 9.28 | 9.51 | 9.25 | 2.77 |
| Av. CPU Usage (%), N3 | 13.33 | 13.13 | 13.10 | 1.74 |
| Av. Power Usage (W), N1 | 3.41 | 3.39 | 3.34 | 2.07 |
| Av. Power Usage (W), N2 | 2.89 | 2.87 | 2.86 | 1.04 |
| Av. Power Usage (W), N3 | 3.53 | 3.44 | 3.39 | 4.05 |

Like the results from trial one, the results of the repeated runs of trial two were also considered to be precise, with the maximum difference being 4.05% for the power usage measured for Node 3. This corresponded to an actual maximum difference of 0.14 Watts which was an acceptable difference in measurements recorded for the same value. All other measurements had a percent difference of less than 4.05% again indicating that they were in an acceptable range of one another.

As the results of all runs for both trial one and trial two were similar to one another it was determined that the experimental setup and methods were robust

enough to perform only one run for all subsequent trials. As such, all other trials for all ReplicaSet values were run only once.

## 5.2 Performance Results

These metrics are of interest as they give insight into the overall performance of the server. The higher the transfer rate, the more efficiently the server is handling requests. The complete versus failed requests give insight into how many requests the server can handle, while the number of non-2xx-responses can tell how many valid responses the server returned. 404 Not Found and 502 Bad Gateway errors are examples of responses that would be considered non-2xx responses. Apache Benchmark classifies all 3xx, 4xx, and 5xx response codes in this category.

### 5.2.1 Complete & Failed Requests

For all 63 trials that were performed on the Drupal server, Apache Benchmark recorded all of the responses with the 200 OK status code, indicating that all the HTTP requests had succeeded. There were no requests for any trials that were reported as having failed. Consequently, there were no non-2xx responses recorded.

This contrasts with the WordPress server results, as approximately 50% of the requests sent to it were considered to have failed with the 502 Bad Gateway status code. In terms of setup, the two applications were deployed similarly via YAML manifests and both used a compatible MariaDB database. The dis-

crepancy in responses is thus rather striking. When querying the k3s cluster for information about the pod in which the WordPress server was deployed, responses always indicated that the pod was running without issue. The logs for the pod indicated the same. Figure 5.1 shows a screenshot of the status of both the WordPress and its associated MariaDB database.

```
root@node-dev-01:/home/pi# kubectl get pods -n wordpress -o wide
NAME                              READY   STATUS    RESTARTS   AGE     IP           NODE
wordpress-668b6f77c6-9twrk        1/1     Running   0          6d23h   10.42.3.48   node-dev-04
wordpress-mariadb-57476fb84-l8tnn   1/1     Running   0          6d23h   10.42.1.54   node-dev-03
root@node-dev-01:/home/pi#
```

**Figure 5.1:** Status of WordPress Pods.

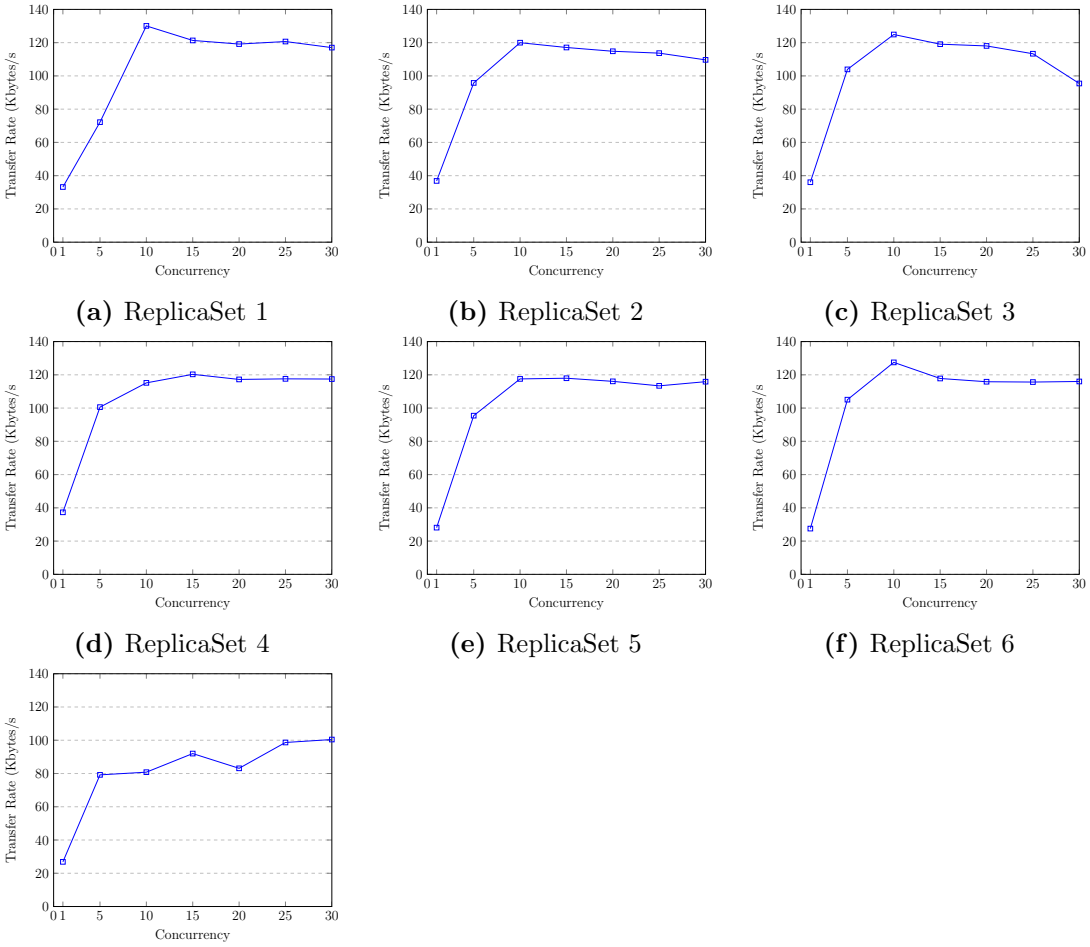
The figure shows all containers deployed in the two pods as ready and their status as running. It also indicates that neither of the pods had to restart since their creation. This shows that the issues with the stability of the server were at the application layer of the web server application itself, rather than at the k3s deployment level. The most likely explanation for this instability is an issue with the Docker Hub image the container application was built from.

Thus the lack of failed requests across the Drupal server tests shows that it was stable and capable of serving all requests at all concurrency levels.

### 5.2.2 Transfer Rate

The Transfer Rate returned by Apache Benchmark was calculated using Equation 4.2. Figure 5.2 shows the transfer rates measured by Apache Benchmark for all concurrency levels and for each ReplicaSet

As Figure 5.2 shows, the trends for each ReplicaSet were comparable to one another. For each trial, the lowest transfer rate occurred at concurrency level




**(a)** ReplicaSet 1 **(b)** ReplicaSet 2 **(c)** ReplicaSet 3

**(d)** ReplicaSet 4 **(e)** ReplicaSet 5 **(f)** ReplicaSet 6

**(g)** ReplicaSet 7

**Figure 5.2:** Transfer Rate vs. Concurrent Requests (All Replica Sets).

one while the highest transfer rate occurred between concurrency levels 10 and 15. The peak transfer rate measured was 130.07 Kbytes/s as shown in Figure 5.2(a). This peak value occurred at concurrency level 10. The next highest transfer rate measured is shown in Figure 5.2(f) and was 127.48 KBytes/s. This rate also occurred when the ReplicaSet was equal to 6 and the concurrency was set to 10. Beyond a concurrency level of 10 the transfer rate began to plateau, with the average rate past that point being around 120 Kbytes/s for all concurrency levels and ReplicaSets. The exception was ReplicaSet 7 which, as Figure 5.2(g) shows, had a maximum transfer rate of 100.43 Kbytes/s. These results indicate that, as the number of concurrent requests increased beyond 10, the ability of the server to handle those requests decreased.

Similar trends were observed for the endurance runs, the results which are shown in Figure 5.3.

Again the results for these trials were consistent across the various ReplicaSets, with the average transfer rate at approximately 120 KBytes/s. The concurrency level for the endurance trials was set at 15 and the number of requests was set such that the trials lasted for 15 minutes. The transfer rate obtained for these trials was similar to those obtained for the trials shown in Figure 5.2, indicating that the concurrency level, not the number of requests or trial time was the determining factor in the transfer rate of the server.

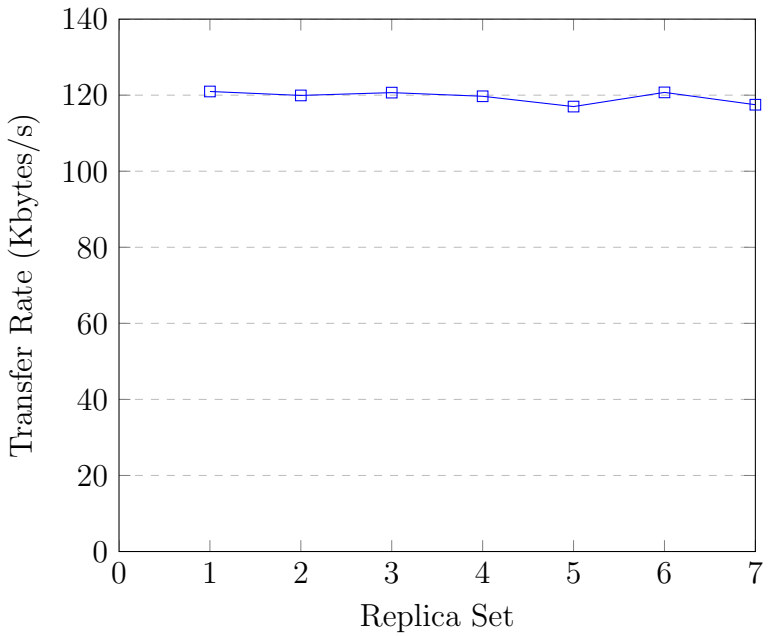



**Figure 5.3:** Transfer Rate vs. Replica Set (Endurance Run, 15 Concurrent Requests).

### 5.2.3 Time Per Request

The Apache Benchmark results obtained for time per request were calculated using Equation 4.1. This metric is similar to the transfer rate metric as both are calculated using the *timetaken* parameter. The primary difference is that the lower the TPR time the more efficiently the server operates. Figure 5.4 shows the results obtained for this metric. The trends observed across the ReplicaSets were similar for this metric, so only the results for ReplicaSet 1 are shown.

Again, the data show that the optimal time per request occurred at a concurrency level of 10, where a TPR of 69.68 milliseconds was measured. Beyond this concurrency level, the time per request plateaued at an average of 75 ms for all

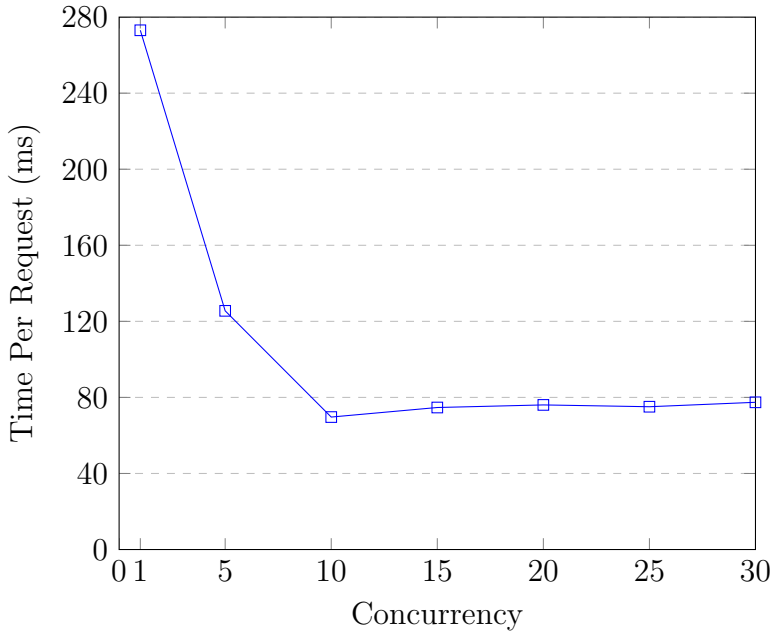



**Figure 5.4:** Time Per Request vs. Concurrent Requests (ReplicaSet 1).

subsequent trials. Like the transfer rate, these results also support the conclusion that concurrency was the determining factor in the efficiency of the web server.

### 5.3 CPU Usage

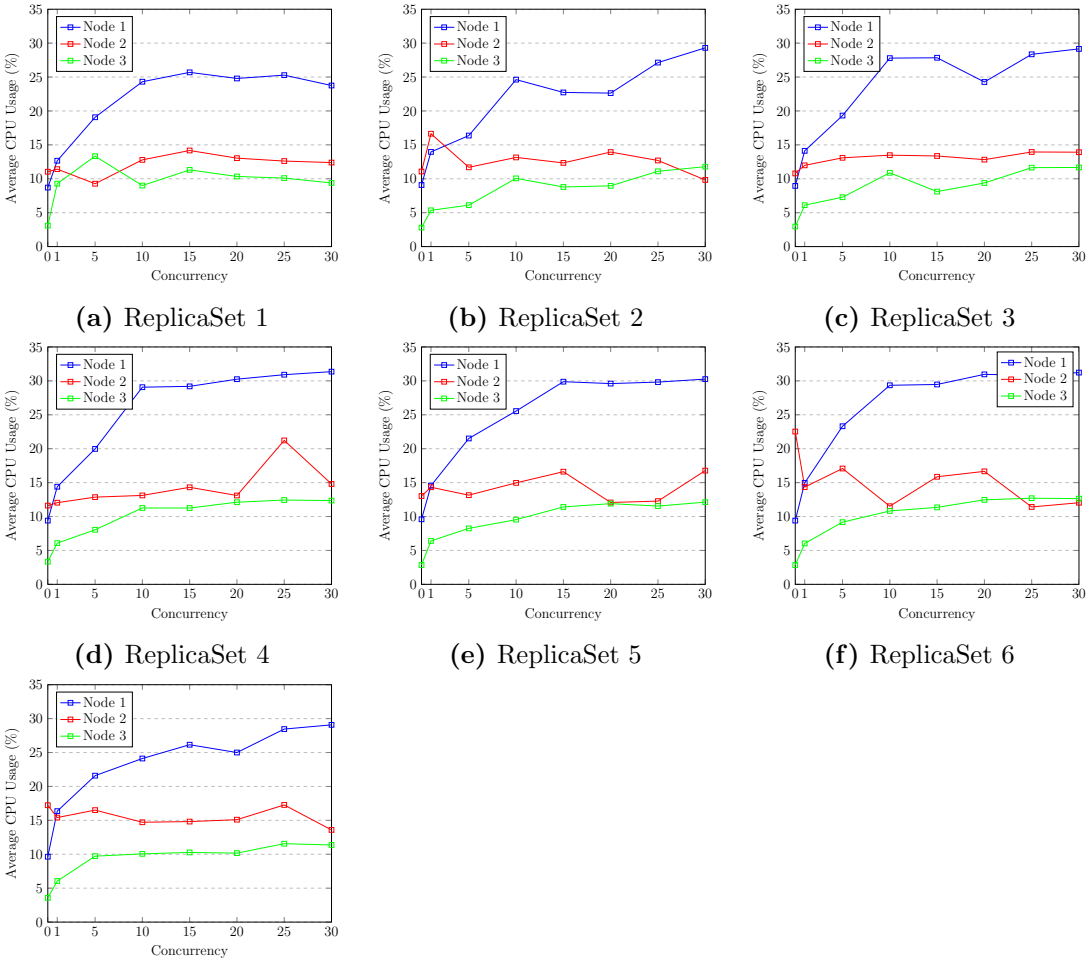
The CPU usage indicates the percentage of the total CPU processing power used to perform the tasks running on the computer at that time. CPU usage provides insight into the performance of the hardware itself, as the lower the CPU usage the more capacity the device has to perform additional tasks. In this research, each node in the k3s cluster had a different set of applications running on them as detailed in Figure 3.1. As such, the CPU usage of each node was different from the other nodes in the cluster. The CPU usage was obtained via Grafana

and was collected for all 63 trials performed. Figure 5.5 shows the average CPU usage of all three nodes for each trial.

The data in Figure 5.5 show that, when compared to one another, the CPU usage of Node 1 is greater than Node 2 which is, in turn, greater than Node 3. This is the case across all ReplicaSets.

### 5.3.1 Node 1

Node 1 was hosting the control plane for the cluster and the pods for the Drupal web server. As such whenever a load was induced on the web server by the Apache Benchmark tool, it had the largest increase in CPU usage. This increase in the CPU usage corresponded to the increase in the number of concurrent requests made to the server. Across all ReplicaSets the CPU usage increased with the increase in concurrency before plateauing between 10 and 15 concurrent requests. On average the node consumed the least amount of CPU resources when there was only one pod deployed for the web server. With only one pod the CPU usage plateaued at an average of 25% while for subsequently deployed pods the CPU usage plateaued at 30% on average. There was not a noticeable increase in CPU usage between the deployment of 4 and 6 replica pods on Node 1. For these trials the 30% plateau was reached between 10 and 15 concurrent requests and remained around that point as the concurrency level increased.




**(a)** ReplicaSet 1 **(b)** ReplicaSet 2 **(c)** ReplicaSet 3

**(d)** ReplicaSet 4 **(e)** ReplicaSet 5 **(f)** ReplicaSet 6

**(g)** ReplicaSet 7

**Figure 5.5:** Average CPU Usage vs. Concurrent Requests (All ReplicaSets).

### 5.3.2 Node 2

Node 2 was hosting the monitoring stack of Prometheus and Grafana. Figure 5.5 shows that when there was no load on the web server at concurrency level zero, Node 2 has a slightly greater CPU usage than Node 1. At the baseline value Node 1 averaged a CPU usage value of 9% while Node 2 averaged a value of 11%. This phenomenon was expected as the monitoring stack on Node 2 was constantly scraping all nodes in the cluster for data while Node 1 was not. There are, however, spikes in CPU usage that seemingly occur at random across the data sets. This is not random, however, but a result of the alert manager that is part of the Grafana application. These spikes in CPU usage Node 2 experiences correspond to the alert manager being updated. As Figure 5.6 shows these spikes occurred regularly at 3-minute intervals. Figure 5.6 was captured from the

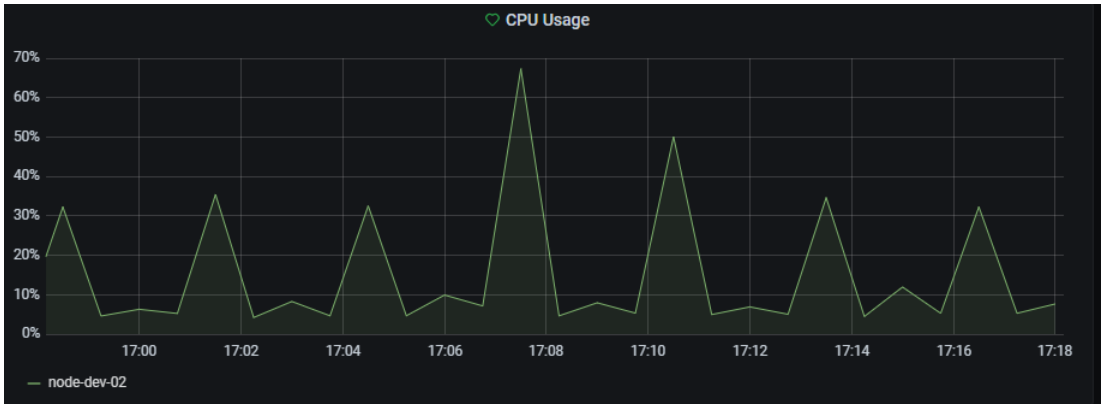



**Figure 5.6:** CPU Usage vs. Time in Minutes (Node 2).

Grafana dashboard hosted on Node 2 and shows the usage spikes occurring at regular three-minute intervals. The seemingly random nature of the CPU spikes

of Node 2 shown in Figure 5.5 was due to one of these spikes occurring during that specific trial. Trials were not structured to be performed around these spikes which is why they appear to be random in Figure 5.5. These spikes notwithstanding, the average CPU usage was around 15% for Node 2 across all the trials. All of the load on Node 2 was induced by the monitoring stack and was independent of the load created on Node 1 by the Apache Benchmark tool. This is why the CPU usage for this node does not increase with the increase in the concurrency level. The increase in ReplicaSets had no impact on Node 2 either, which is why there is no visible change in the CPU usage of this node between the different ReplicaSets.

### 5.3.3 Node 3

Node 3 hosted the MariaDB database which was the required counterpart of the Drupal web server. Overall this node consumed the least amount of CPU resources with the usage plateauing around 12% on average. As with other measurements, this plateau typically occurred once the number of concurrent requests reached 10 to 15. As the database was linked to the web server it made sense that it would see an increase in utilization whenever the load on the web server increased. It also made sense that this increase would be minimal when compared to the increase experienced by Node 1 as the Apache Benchmark tool makes HTTP requests to the web server and does not write changes to the database. Like with Node 2, the ReplicaSets had no impact on Node 3 which is why there was no change in the trends for this node between data sets.

### 5.3.4 Endurance Run

Figure 5.7 shows the average CPU usage data obtained for the 15-minute endurance runs.

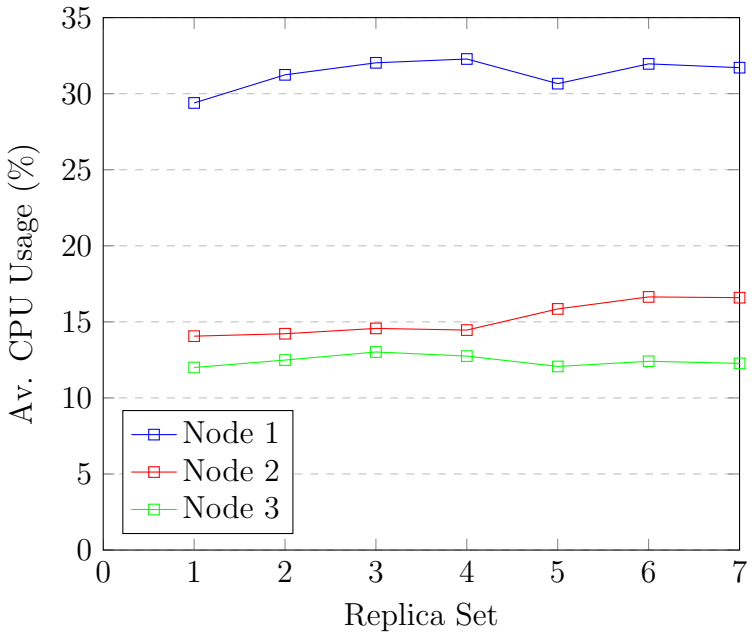



**Figure 5.7:** CPU Usage vs. Replica Set (Endurance Run, 15 Concurrent Requests).

These data correspond closely with those obtained for the other trials with the average usage being approximately 30% for Node 1, 15% for Node 2, and 12% for Node 3. The notable difference here is that, for Node 1, the longer trials resulted in a higher CPU usage overall. For example, Figure 5.5(a) shows that the average CPU usage was around 25% when 15 concurrent requests were made during the two-minute trials. However, when the trial time increased to 15 minutes the average increased to 30%. This observation has two possible explanations. The first possibility is that because more data points were collected

during the endurance trial, the average value obtained from these data points is more accurate. The other possibility is that a sustained load on the web server results in another factor, like the temperature of the Raspberry Pi, increasing. This increase in temperature could cause a related increase in the utilization of the CPU. This assertion is just a theory however as temperature measurements were not taken as part of this research.

## 5.4 Power Consumption

The final metric of interest researched for this thesis was power consumption. This metric was of interest because many IoT devices operate off battery power, so power consumption must be taken into account. If the power consumption is too high, then the device battery could drain quickly and the device could, at best, need to be recharged, or at worst, have to be replaced within a short time frame. Neither of these situations is ideal, and the power consumption of the device should always be minimized, especially when battery-powered. Power consumption measurements were taken 10 times a second and were measured in Watts by the WaveShare Power Monitor HAT.

### 5.4.1 Baseline Power Measurements

Before conducting any trials baseline power measurements were taken for all three Raspberry Pi boards in various states. The device states were as follows:

- **State 1:** Operating system only. No additional applications or tools were deployed on Raspberry Pis.

- **State 2:** k3s cluster deployed. No additional applications were deployed within the cluster.
- **State 3:** Monitoring stack deployed. Grafana and Prometheus deployed on Node 2. Prometheus Node Exporter deployed on all nodes.

These three states were chosen to provide insight into the relative increases in power consumption caused by deploying various parts of the k3s cluster. All baseline measurements were taken for 2 minutes at a rate of 10 measurements per second for all three devices. Other than the applications listed for each state the devices were idle. Table 5.3 shows the average of these measurements.

**Table 5.3:** Baseline Average Power Consumption in Watts.

| **State** | **Node 1** | **Node 2** | **Node 3** |
|---|---|---|---|
| 1 | 2.09 | 2.10 | 2.03 |
| 2 | 2.88 | 2.52 | 2.45 |
| 3 | 2.97 | 2.80 | 2.58 |

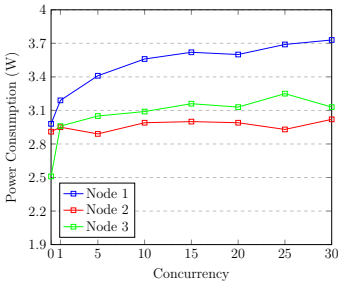
In state One all three of the devices measured a baseline power consumption of around 2.00 Watts. When the k3s cluster was deployed the power consumption increased to 2.88, 2.52, and 2.45 Watts for each node, respectively. Node 1 had the largest increase in power use as it was hosting the control plane for the cluster while the agent nodes each experienced an increase of 0.42W. Between

State Two and State Three Node 2 experienced the largest increase in power consumption due to the deployment of the monitoring stack. The smaller increases in power on Node 1 and Node 3 were due to the Prometheus node exporter pods that were deployed on each of them.

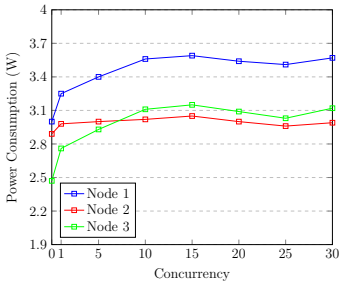
### 5.4.2 Power Consumption Measurements Under Load

The power consumption data were obtained via the power monitors and was collected for all 63 trials performed. Figure 5.8 shows the average power consumption of all three nodes for each trial.

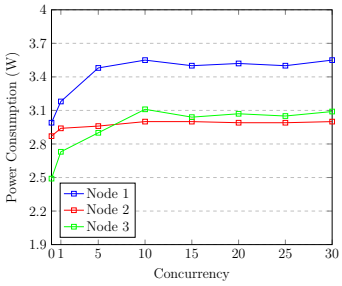
As with the data sets for the performance metrics and CPU usage, the power consumption data sets display similar trends. The power consumption of Node 1 increases until a concurrency level of 10 to 15 is reached at which point it plateaus at approximately 3.6 Watts. The power used by Node 2 remains fixed at around 2.9W regardless of the trial conducted or the number of ReplicaSets deployed. Like the data in Figure 5.5 deviations from this average were attributed to Grafana updating its alert manager every three minutes. The results for Node 3 proved to be the most interesting as it had the lowest baseline power consumption of the three nodes. However, once load was induced on the web server, the power consumption of Node 3 increased until it equaled or exceeded that of Node 2. Node 3 had an average power consumption of 3.1W across the trials, with the power consumption plateauing for each at a concurrency level of 10. Taking the CPU usage data as a reference, the expectation would have been that Node 2 would have had a higher average power consumption when compared to Node 3.



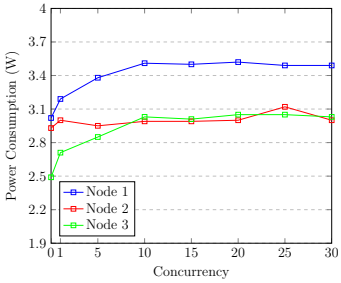

**(a)** ReplicaSet 1



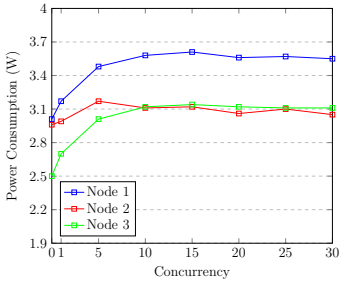

**(b)** ReplicaSet 2



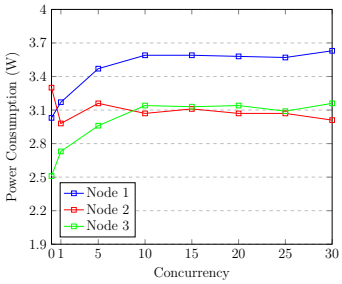

**(c)** ReplicaSet 3



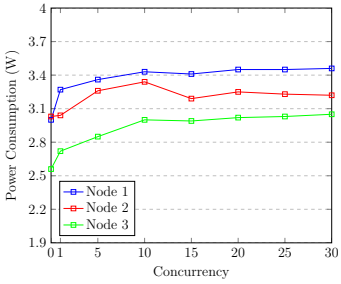

**(d)** ReplicaSet 4




**(e)** ReplicaSet 5




**(f)** ReplicaSet 6




**(g)** ReplicaSet 7

**Figure 5.8:** Power Consumption vs. Concurrent Requests (All Replica Sets).

## 5.5 Comparison to Local Raspberry Pi Server

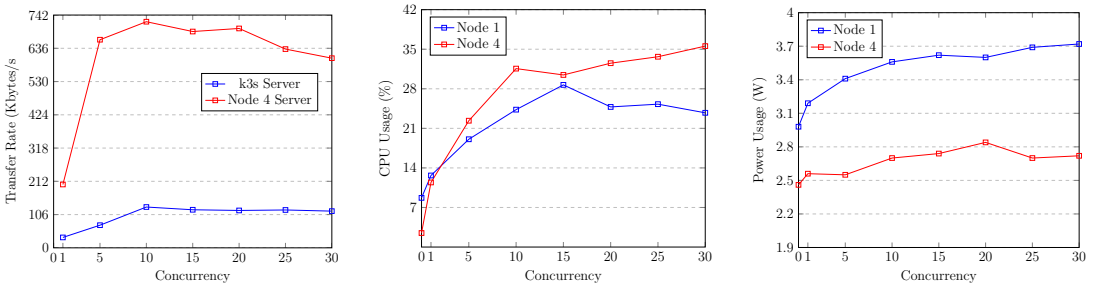
The trial runs defined in Table 4.1 were performed on the Drupal server running locally on the fourth Raspberry Pi and performance, CPU Usage, and Power Consumption data were collected. For the purposes of this discussion it will be referred to as Node 4. These results were then compared to the ReplicaSet 1 data from Node 1 of the k3s server trials. The ReplicaSet 1 data were chosen as they were the set that yielded the best performance for the lowest resource consumption of all the cloud-based data gathered. Table 5.4 shows the baseline data gathered for Node 4.

**Table 5.4:** Baseline Measurements for Node 4.

| **Metric** | **Baseline Measurement** |
|---|---|
| CPU Usage (%) | 2.48 |
| Power Usage, No Server (W) | 2.13 |
| Power Usage, Server Deployed (W) | 2.46 |

The trials were run for the two servers and the results were compared. Figure 5.9 shows the results for all the metrics of interest. In Figure 5.9(b) and Figure 5.9(c) only the data obtained for Node 1 are shown as this was the node running the Drupal server.

As the data in Figure 5.9(a) show the transfer rate of the Node 4 server was significantly faster than that of the k3s-based server. Both servers achieved their




**(a)** Transfer Rate vs. Concurrency

**(b)** CPU Usage vs. Concurrency

**(c)** Power Consumption vs. Concurrency

**Figure 5.9:** Metrics Comparison, Node 4 and k3s Servers.

maximum transfer rates at 10 concurrent requests, with the k3s server reaching a rate of 130.07 KBytes/s and the Node 4 server reaching 720.74 KBytes/s as reported by Apache Benchmark. The Node 4 server achieved a transfer rate of approximately 5.5 times greater than that of the k3s server. This differential in performance was maintained overall as the concurrency increased with the performance of both servers falling to 117.02 and 604.53 KBytes/s at concurrency level 30, respectively. While the Node 4 server achieved a higher transfer rate, it still maintained the same trend observed in all the other tests with the performance peaking between 10 and 15 concurrent requests and generally plateauing/falling thereafter. The decrease in performance between the two servers at 30 concurrent requests was more pronounced for the Node 4 server compared to the k3s server with it falling 19.53% percent from its peak compared to 10.53% for the K3s server.

Comparison of the endurance runs showed similar results with the Node 4 server achieving a 6-fold increase in speed over the k3s server at 737.51 and 120.97 Kbytes/s, respectively. These 12,000 requests sent 15 at a time took the

k3s server 14 minutes and 58 seconds to process while it took the Node 4 server 2 minutes and 29 seconds. It was also observed that, like the k3s server, the Node 4 server completed all requests successfully across all of its runs.

Figure 5.9(b) shows the CPU usage for the two devices. As the figure shows the CPU consumption was similar. At all but the lowest concurrency levels the Node 4 server had a higher CPU usage than Node 1, though the usage never exceeded 35% at its peak. The higher utilization was not surprising given Node 4's better performance in serving requests compared to Node 1. At a similar transfer rate, it would be expected that Node 1 would also experience a higher level of CPU usage likely exceeding that of the other Node 4 due to it also running the k3s control plane. These results were comparable to those achieved by Khanjan *et al.* [17] in their research on Raspberry Pi CPU usage, where theirs peaked at approximately 37% when under load from Apache Benchmark.

Figure 5.9(c) shows the power consumption for the two devices. As the figure shows, Node 1 consumed more power than Node 4 during all trials. The difference in power consumption was 1 Watt on average with the highest consumption for Node 1 being 3.75 Watts and that of the Node 4 being 2.74 Watts. This was again expected as Node 1 was also running the k3s control plane while Node 4 was only running the Drupal server. These results were similar to those obtained by Bekaroo and Santokhee [18] where Node 1 had comparable power use to their Raspberry Pi when performing online tasks and Node 4 comparable use to their Raspberry Pi when performing offline tasks.

Overall the performance of the Node 4-based server exceeded that of the k3s server for each trial. The difference in power consumption and CPU usage between the devices was minimal and more than made up for by the performance of the Node 4 server in handling requests. The reason for the large difference in performance was attributed to the requests having to be routed through the k3s cluster to the pod running the Drupal server on Node 1, versus the requests going directly to the server hosted on Node 4.

In this instance, the primary benefit of using the cloud-based cluster was in its ease of use. Assuming the k3s cluster is already deployed, the Drupal server can be deployed by simply applying the manifest files for the server and its database. This process can be accomplished via two commands in the terminal and takes less than 5 minutes from deployment to running. Deployment of Apache, PHP, MariaDB, and Drupal services is accomplished for the user through these commands alone. The server can also be taken offline just as quickly. In contrast, each of these services had to be installed and configured separately for the Node 4 server and care had to be taken that all versions of these tools were compatible with the Raspberry Pi's ARM processor. While not overly complicated, this process was much more involved than simply typing in two commands at the terminal. Decommissioning the server was also more complex, with each service having to be stopped separately and the tools each manually removed from the Raspberry Pi. Thus the main trade-off between the two servers was performance versus ease of deployment and maintenance.

# Chapter 6. Conclusion and Future Work

## 6.1 Concluding Remarks

This research sought to deploy a web server in a lightweight cloud computing infrastructure on a small, IoT-type device and characterize it. The metrics selected to characterize the web server and cluster were the performance of the server as determined by Apache Benchmark, the CPU usage of each device in the cluster, and the power consumption of all devices. K3s was selected as the cloud infrastructure for the research and Raspberry Pi 4 Model B's were used as the hardware. Experiments were performed on both a web server hosted within the k3s cluster and on a server hosted locally on a separate Raspberry Pi board.

Results of the experiment showed that the overall performance of the cluster-based server was low compared to that of the locally hosted server. While the power consumption and CPU usage were comparable between the two, the transfer rate of the cluster-based server was six to seven times less than that of the other. The power consumption and CPU usage data obtained for all experiments were comparable to those found in related research found in other publications. Additionally configuring the server to be part of a Kubernetes Deployment with scalable ReplicaSets provided no additional benefit, with the performance of the server being comparable or worse as the number of ReplicaSets increased. This

indicates that additional pods in a deployment only provide a benefit when the existing pods are under substantial load and at risk of reaching their maximum resource usage.

The benefit of using a cloud-based solution at this small scale was determined to primarily lie in its ease of implementation and its distributed nature, with the server able to be deployed in the cluster with a few basic commands and automatically maintaining the minimum number of pods specificed by the user.

## 6.2 Future Work

This research was a preliminary study into characterizing a cloud-based computing cluster in an IoT environment and thus the scope was somewhat limited. One of the primary limitations was the lack of full compatibility that many applications have with ARM devices. This reality prevented the use of certain tools for load testing like Locust or features of k3s like the automatic scaling of pods in a deployment. Future endeavors based on this research would seek to use platforms more compatible with these applications and tools or build custom versions of the tools that could be run on ARM devices.

Optimization of the web server services could also yield interesting results. In this research, the Drupal server was left "as is" thus representing the baseline of what is possible for a cloud-based web server. Repeating the experiment with a focus on optimizing the web server could result in improved performance when compared to the natively hosted server.

Another extension of this work would be to measure additional metrics such as CPU temperature to see how they impact the performance of the cluster and its applications. Temperature has a significant impact on computers and increases beyond what they are rated for can lead to performance and stability issues. It would be interesting to see the impact that running a lightweight cluster would have on the temperature of the Raspberry Pi CPU.

Lastly deploying a wider variety of applications on the k3s cluster would be another extension of this research. The web server was selected as it was relatively simple to deploy and is a common object deployed in the cloud, but the variety of applications that can be deployed in a cloud context is vast. Deploying a wide variety of these applications and seeing which ones are best suited to an IoT-type environment would be interesting as well.

# References


[1] "What is cloud computing? — microsoft azure." https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing. [Accessed: 13-June-2024].

[2] "Gartner says worldwide iaas public cloud services market grew 31 percent in 2016." https://www.gartner.com/en/newsroom/press-releases/2017-09-27-gartner-says-worldwide-iaas-public-cloud-services-market-grew-31-percent-in-2016: :text=Analysts [Accessed: 06-January-2024].

[3] R. Straits, "Cloud computing market size, share growth report 2031." https://straitsresearch.com/report/cloud-computing-market. [Accessed: 06-January-2024].

[4] "60% of business data is kept on the cloud, leading to a cagr of 30.8% for the cloud encryption market." https://alltechmagazine.com/60-of-business-data-is-kept-on-the-cloud/, March 2023. [Accessed: 06-January-2024].

[5] J. McKendrick, "With internet of things and big data, 92% of everything we do will be in the cloud." https://www.forbes.com/sites/joemckendrick/2016/11/13/with-internet-of-things-and-big-data-92-of-everything-we-do-will-be-in-the-cloud/, November 2016. [Accessed: 06-January-2024].

[6] RIAA, "U.s. music revenue database." https://www.riaa.com/u-s-sales-database/. [Accessed: 06-January-2024].

[7] S. Thomas, "Cloud computing is the engine powering generative ai." https://individuals.voya.com/insights/investment-insights/cloud-computing-engine-powering-generative-ai, September 2023. [Accessed: 23-May-2024].

[8] B. P. Gajendra and V. K. Singh, "Achieving cloud security using third party auditor, md5 and identity-based encryption," in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pp. 1304–1309, IEEE, 2016.

[9] "What is a container?." https://www.docker.com/resources/what-container/. [Accessed: 06-January-2024].

[10] "Guide to containers." https://www.atatus.com/ask/guide-to-containers. [Accessed: 07-January-2024].

[11] "What is kubernetes?." https://cloud.google.com/learn/what-is-kubernetes. [Accessed: 07-January-2024].

[12] "Kubernetes components." https://kubernetes.io/docs/concepts/overview/components/. [Accessed: 07-January-2024].

[13] M. Fogli, T. Kudla, B. Musters, G. Pingen, C. Van den Broek, H. Bastiaansen, N. Suri, and S. Webb, "Performance evaluation of kubernetes distributions (k8s, k3s, kubeedge) in an adaptive and federated cloud infrastructure for disadvantaged tactical networks," in *2021 International Conference on Military Communication and Information Systems (ICMCIS)*, pp. 1–7, IEEE, 2021.

[14] S. Böhm and G. Wirtz, "Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes.," in *ZEUS*, pp. 65–73, 2021.

[15] A. Kivimäki, "Evaluation of lightweight kubernetes distributions in edge computing context," Master's thesis, 2021.

[16] C. Yström and A. Stenborg, "Performance comparison between a kubernetes cluster and an embedded system," 2021.

[17] V. Khanjan, V. Patel, and M. Gangrade, "Comparison between performance of raspberry pi 4b and laptop computer as a server," *International Research Journal of Engineering and Technology*, vol. 08, pp. 696–701, Nov. 2021.

[18] G. Bekaroo and A. Santokhee, "Power consumption of the raspberry pi: A comparative analysis," in *2016 IEEE International Conference on Emerging Technologies and Innovative Business Practices for the Transformation of Societies (EmergiTech)*, pp. 361–366, 2016.

[19] Q. He, B. Segee, and V. Weaver, "Raspberry pi 2 b+ gpu power, performance, and energy implications," in *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 163–167, 2016.

[20] "K3s - lightweight kubernetes — k3s." https://docs.k3s.io/, February 2024. [Accessed: 03-February-2024].

[21] "Certified kubernetes software conformance." https://www.cncf.io/training/certification/software-conformance/. [Accessed: 03-February-2024].

[22] "K3s vs k8s – what's the difference between k8s and k3s." https://www.civo.com/blog/k8s-vs-k3s. [Accessed: 03-February-2024].

[23] "Requirements — k3s." https://docs.k3s.io/installation/requirements, February 2024. [Accessed: 03-February-2024].

[24] "Architecture — k3s." https://docs.k3s.io/architecture, February 2024. [Accessed: 03-February-2024].

[25] P. Fromaget, "The epic story of the raspberry pi." https://raspberrytips.com/raspberry-pi-history/, April 2021. [Accessed: 08-February-2024].

[26] R. P. Ltd, "Buy a raspberry pi 4 model b." https://www.raspberrypi.com/products/raspberry-pi-4-model-b/. [Accessed: 08-February-2024].

[27] "Automate everything - ansible quick start guide [book]." https://www.oreilly.com/library/view/ansible-quick-start/9781789532937/01c6351e-bb66-4de0-832a-8999fa6c724e.xhtml. [Accessed: 03-February-2024].

[28] V. Rabah, "Build a kubernetes cluster using k3s via ansible." https://github.com/k3s-io/k3s-ansible. [Accessed: 03-February-2024].

[29] Prometheus, "Overview — prometheus." https://prometheus.io/docs/introduction/overview/. [Accessed: 03-February-2024].

[30] "Get started with grafana and prometheus — grafana documentation." https://grafana.com/docs/grafana/latest/getting-started/get-started-grafana-prometheus/. [Accessed: 13-June-2024].

[31] "Grafana — query, visualize, alerting observability platform." https://grafana.com/grafana/, [Accessed: 03-February-2024].

[32] S.-A. Segun, "Grafana  prometheus kubernetes cluster monitoring." https://earthly.dev/blog/grafana-and-prometheus-k8s/, August 2022. [Accessed: 03-February-2024].

[33] "4-ch current/voltage/power monitor hat for raspberry pi, i2c/smbus interface." https://www.waveshare.com/current-power-monitor-hat.htm. [Accessed: 09-February-2024].

[34] "About." https://www.drupal.org/about, October 2015. [Accessed: 13-June-2024].

[35] "About mariadb server." https://mariadb.org/about/, October 2015. [Accessed: 13-June-2024].

[36] "Apache http server benchmark - openbenchmarking.org." https://openbenchmarking.org/test/pts/apache&eval=62031bd021ebd0f0bfdaac7053e9bdfb580842b3#metrics. [Accessed: 23-May-2024].

[37] "ab - apache http server benchmarking tool - apache http server version 2.4." https://httpd.apache.org/docs/2.4/programs/ab.html. [Accessed: 05-February-2024].

[38] "Objects in kubernetes." https://kubernetes.io/docs/concepts/overview/working-with-objects/. [Accessed: 05-February-2024].

[39] "carlosedp - overview." https://github.com/carlosedp. [Accessed: 05-February-2024].

[40] C. Eduardo, "Cluster monitoring stack for clusters based on prometheus operator." https://github.com/carlosedp/cluster-monitoring. [Accessed: 05-February-2024].

# Appendix A. Installing & Setting up Raspberry Pi OS on a Raspberry Pi 4

1. Browse to https://www.raspberrypi.com/software/operating-systems/ and select the version of R Pi OS you wish to run. In this example the latest 64-bit version of the R Pi OS was used (Debian 11, Kernel 6.1, release date 10/30/2021).

2. Install the Raspberry Pi imager from https://www.raspberrypi.com/software/ (v1.7.5 used for this walkthrough).

3. Insert a microSD card into your computer.

4. Launch the Raspberry Pi Imager Software. For the "Choose OS" option go to "Use Custom" and select the RPi OS image you downloaded in step 1a. For the "Choose Storage" box select the microSD card you just inserted into your computer.

5. Select the "Write" button and wait for the image to be written to the SD card. This may take some time. (WARNING: performing this write operation will delete all other files from the SD card. Make sure you remove any files you want to keep from the SD card prior to write operation).

6. Once the OS image has been written, remove the SD card from your computer and insert it into the microSD card slot on the RPi.

7. Connect your monitor, keyboard, and mouse to the RPi.

8. Turn on the RPi and let the boot process run. This may take some time.

9. Once booted, go through the prompts to set up OS. Make sure that you connect your RPis to a network as internet access will be required to complete the setup. (IMPORTANT: if you are using an older version of RPi OS press "Skip" when prompted to do the software update check).

10. Once the initial set up is complete open the terminal and enter "sudo raspi-config" and press enter.

11. From the menu that pops up go to System Options > Hostname and enter the desired hostname for your device. Press OK.

12. Navigate to Interface Options > SSH > Yes to enable the SSH server. Select finish on the raspi-config menu and reboot the device.

13. Once rebooted using the "sudo" command open the file "/boot/cmdline.txt" using a text editor and append the following to the existing line in the file (do not add a new line): cgroup_enable=cpuset cgroup_memory=1 cgroup_enable=memory. Save and close the cmdline.txt file. Reboot the device.

14. Repeat steps 3 through 13 for all Raspberry Pi devices.

# Appendix B. Installing Ansible & Using it to Setup an N-Node k3s Cluster

1. Set up headless SSH access on all devices that will act as nodes in the cluster.

2. Ensure root privileges are enabled on all devices that will act as nodes in the cluster.

3. To download Ansible we must first install pipx. Pipx can be installed using the following command "python3 -m pip install −−user pipx".

4. Pipx installs in .local/bin. We want to add this location to the PATH permanently. To do this we need to edit the file .bashrc in the home directory.

5. Open .bashrc in a text editor and add the command "export PATH= /.local/bin:$PATH" to the very end of the file. Save the file and close it.

6. Depending on the Linux distribution you are using you may or may not need to install the python3 virtual environment in order to install Ansible. If you do not already have the python3 virtual environment you can install it by running the command "sudo apt install python3-venv". If you already have the virtual environment installed you may skip this step.

7. To install Ansible run the command “pipx install −−include-deps ansible” from the Linux terminal. If you want to install something other than the latest full version of Ansible please consult the Ansible installation guide.

8. Download or clone the k3s-ansible repository from https://github.com/rancher/k3s-ansible .

9. Navigate to your downloads folder.

10. If you downloaded the file via a web browser, extract the source files from the .zip using the command “unzip k3s-ansible-master k3s-ansible-master.zip”

11. Run the command “cd k3s-ansible-master/inventory”

12. Copy the contents of the sample directory to your working directory by using the command “cp -R inventory/sample inventory/<directory name>”.

13. Navigate to this newly created directory and open the file hosts.ini. Change the IP addresses listed in the file to the IP addresses for your RPi devices. Add the IP for the master device under the [master] heading and the IP address for your node(s) under the [node] heading.

14. Edit the file “inventory/group_vars/all.yml” by changing ansible_user to pi.

15. Navigate back to the k3s-ansible-master directory and run the command “ansible-playbook site.yml -i inventory/<directory name>/hosts.ini” and wait for the setup to complete. If the setup was successful you should see failed=0 in the PLAY RECAP for all devices in the cluster.

**Additional Information:**

- If you brick your cluster somehow and want to start over you can reset it using the command “ansible-playbook reset.yml -i inventory/hosts.ini”.
- To safely shut down your RPi cluster without having to log into each Pi device, run the command “ansible all -i inventory/hosts.ini -a ”shutdown now” -b”. It will take a minute or so to shutdown the devices at which point you should be able to unplug them safely.

# Appendix C. Python Source Code ina219.py

```
import smbus
import time
import csv

# Config Register (R/W)
_REG_CONFIG                 = 0x00
# SHUNT VOLTAGE REGISTER (R)
_REG_SHUNTVOLTAGE           = 0x01

# BUS VOLTAGE REGISTER (R)
_REG_BUSVOLTAGE             = 0x02

# POWER REGISTER (R)
_REG_POWER                  = 0x03

# CURRENT REGISTER (R)
_REG_CURRENT                = 0x04

# CALIBRATION REGISTER (R/W)
_REG_CALIBRATION            = 0x05

class BusVoltageRange:
    """Constants for ``bus_voltage_range``"""
    RANGE_16V               = 0x00      # set bus voltage range to 16V
    RANGE_32V               = 0x01      # set bus voltage range to 32V (default)

class Gain:
    """Constants for ``gain``"""
    DIV_1_40MV              = 0x00      # shunt prog. gain set to  1, 40 mV range
    DIV_2_80MV              = 0x01      # shunt prog. gain set to /2, 80 mV range
    DIV_4_160MV             = 0x02      # shunt prog. gain set to /4, 160 mV range
```

```
    DIV_8_320MV             = 0x03      # shunt prog. gain set to /8, 320 mV range

class ADCResolution:
    """Constants for ``bus_adc_resolution`` or ``shunt_adc_resolution``"""
    ADCRES_9BIT_1S          = 0x00      #  9bit,   1 sample,     84us
    ADCRES_10BIT_1S         = 0x01      # 10bit,   1 sample,    148us
    ADCRES_11BIT_1S         = 0x02      # 11 bit,  1 sample,    276us
    ADCRES_12BIT_1S         = 0x03      # 12 bit,  1 sample,    532us
    ADCRES_12BIT_2S         = 0x09      # 12 bit,  2 samples,  1.06ms
    ADCRES_12BIT_4S         = 0x0A      # 12 bit,  4 samples,  2.13ms
    ADCRES_12BIT_8S         = 0x0B      # 12bit,   8 samples,  4.26ms
    ADCRES_12BIT_16S        = 0x0C      # 12bit,  16 samples,  8.51ms
    ADCRES_12BIT_32S        = 0x0D      # 12bit,  32 samples, 17.02ms
    ADCRES_12BIT_64S        = 0x0E      # 12bit,  64 samples, 34.05ms
    ADCRES_12BIT_128S       = 0x0F      # 12bit, 128 samples, 68.10ms

class Mode:
    """Constants for ``mode``"""
    POWERDOW                = 0x00      # power down
    SVOLT_TRIGGERED         = 0x01      # shunt voltage triggered
    BVOLT_TRIGGERED         = 0x02      # bus voltage triggered
    SANDBVOLT_TRIGGERED     = 0x03      # shunt and bus voltage triggered
    ADCOFF                  = 0x04      # ADC off
    SVOLT_CONTINUOUS        = 0x05      # shunt voltage continuous
    BVOLT_CONTINUOUS        = 0x06      # bus voltage continuous
    SANDBVOLT_CONTINUOUS    = 0x07      # shunt and bus voltage continuous


class INA219:
    def __init__(self, i2c_bus=1, addr=0x40):
        self.bus = smbus.SMBus(i2c_bus);
        self.addr = addr

        # Set chip to known config values to start
        self._cal_value = 0
```

```
        self._current_lsb = 0
        self._power_lsb = 0
        self.set_calibration_32V_2A()

    def read(self,address):
        data = self.bus.read_i2c_block_data(self.addr, address, 2)
        return ((data[0] * 256 ) + data[1])

    def write(self,address,data):
        temp = [0,0]
        temp[1] = data & 0xFF
        temp[0] =(data & 0xFF00) >> 8
        self.bus.write_i2c_block_data(self.addr,address,temp)

    def set_calibration_32V_2A(self):
        """Configures to INA219 to be able to measure up to 32V and 2A of current. Counter
           overflow occurs at 3.2A.
           ..note :: These calculations assume a 0.1 shunt ohm resistor is present
        """
        # By default we use a pretty huge range for the input voltage,
        # which probably isn't the most appropriate choice for system
        # that don't use a lot of power.  But all of the calculations
        # are shown below if you want to change the settings.  You will
        # also need to change any relevant register settings, such as
        # setting the VBUS_MAX to 16V instead of 32V, etc.

        # VBUS_MAX = 32V             (Assumes 32V, can also be set to 16V)
        # VSHUNT_MAX = 0.32          (Assumes Gain 8, 320mV, can also be 0.16, 0.08, 0.04)
        # RSHUNT = 0.1               (Resistor value in ohms)

        # 1. Determine max possible current
        # MaxPossible_I = VSHUNT_MAX / RSHUNT
        # MaxPossible_I = 3.2A

        # 2. Determine max expected current
```

```
# MaxExpected_I = 2.0A

# 3. Calculate possible range of LSBs (Min = 15-bit, Max = 12-bit)
# MinimumLSB = MaxExpected_I/32767
# MinimumLSB = 0.000061              (61uA per bit)
# MaximumLSB = MaxExpected_I/4096
# MaximumLSB = 0,000488              (488uA per bit)

# 4. Choose an LSB between the min and max values
#    (Preferrably a roundish number close to MinLSB)
# CurrentLSB = 0.0001 (100uA per bit)
self._current_lsb = .1  # Current LSB = 100uA per bit

# 5. Compute the calibration register
# Cal = trunc (0.04096 / (Current_LSB * RSHUNT))
# Cal = 4096 (0x1000)

self._cal_value = 4096

# 6. Calculate the power LSB
# PowerLSB = 20 * CurrentLSB
# PowerLSB = 0.002 (2mW per bit)
self._power_lsb = .002  # Power LSB = 2mW per bit

# 7. Compute the maximum current and shunt voltage values before overflow
#
# Max_Current = Current_LSB * 32767
# Max_Current = 3.2767A before overflow
#
# If Max_Current > Max_Possible_I then
#    Max_Current_Before_Overflow = MaxPossible_I
# Else
#    Max_Current_Before_Overflow = Max_Current
# End If
#
```

```
        # Max_ShuntVoltage = Max_Current_Before_Overflow * RSHUNT
        # Max_ShuntVoltage = 0.32V
        #
        # If Max_ShuntVoltage >= VSHUNT_MAX
        #     Max_ShuntVoltage_Before_Overflow = VSHUNT_MAX
        # Else
        #     Max_ShuntVoltage_Before_Overflow = Max_ShuntVoltage
        # End If

        # 8. Compute the Maximum Power
        # MaximumPower = Max_Current_Before_Overflow * VBUS_MAX
        # MaximumPower = 3.2 * 32V
        # MaximumPower = 102.4W

        # Set Calibration register to 'Cal' calculated above
        self.write(_REG_CALIBRATION, self._cal_value)

        # Set Config register to take into account the settings above
        self.bus_voltage_range = BusVoltageRange.RANGE_32V
        self.gain = Gain.DIV_8_320MV
        self.bus_adc_resolution = ADCResolution.ADCRES_12BIT_32S
        self.shunt_adc_resolution = ADCResolution.ADCRES_12BIT_32S
        self.mode = Mode.SANDBVOLT_CONTINUOUS
        self.config = self.bus_voltage_range << 13 | \
                      self.gain << 11 | \
                      self.bus_adc_resolution << 7 | \
                      self.shunt_adc_resolution << 3 | \
                      self.mode
        self.write(_REG_CONFIG, self.config)

    def getShuntVoltage_mV(self):
        self.write(_REG_CALIBRATION, self._cal_value)
        value = self.read(_REG_SHUNTVOLTAGE)
        if value > 32767:
            value -= 65535
```

```
        return value * 0.01

    def getBusVoltage_V(self):
        self.write(_REG_CALIBRATION, self._cal_value)
        self.read(_REG_BUSVOLTAGE)
        return (self.read(_REG_BUSVOLTAGE) >> 3) * 0.004

    def getCurrent_mA(self):
        value = self.read(_REG_CURRENT)
        if value > 32767:
            value -= 65535
        return value * self._current_lsb

    def getPower_W(self):
        value = self.read(_REG_POWER)
        if value > 32767:
            value -= 65535
        return value * self._power_lsb


if __name__=='__main__':

    ina1 = INA219(addr=0x40)
    ina2 = INA219(addr=0x41)
    ina3 = INA219(addr=0x42)
    ina4 = INA219(addr=0x43)

    print("ina219 test")
    with open ("drupal_n1600_c30_Rpi1_base.csv", mode="w") as csvfile:
        fieldnames =["Timestamp", "Power (V)"]
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        while True:
            # voltage on V- (load side)
            bus_voltage4 = ina4.getBusVoltage_V()
```

```
# voltage between V+ and V- across the shunt
shunt_voltage4 = ina4.getShuntVoltage_mV() / 1000
# current in mA
current4 = ina4.getCurrent_mA()
# power in watts
power4 = ina4.getPower_W()
# current time in H:M:S format
timestamp = time.strftime("%H:%M:%S", time.localtime())

#print("")
writer.writerow({"Timestamp": timestamp, "Power (V)": power4})

#sleep between samples of data
time.sleep(0.1)
```