A Low-Cost Continuous Development Architecture Based on Container Orchestration

Alessandro Caetano Beltrão

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientadores: Guilherme Horta Travassos

Breno Bernard Nicolau de França

Rio de Janeiro

Setembro de 2022

A LOW-COST CONTINUOUS DEVELOPMENT ARCHITECTURE BASED ON CONTAINER ORCHESTRATION

Alessandro Caetano Beltrão

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientadores: Guilherme Horta Travassos

Breno Bernard Nicolau de França

Aprovada por: Prof. Guilherme Horta Travassos

Prof. Breno Bernard Nicolau de França

Prof. Claudio Miceli de Farias

Prof. Lincoln Souza Rocha

RIO DE JANEIRO, RJ - BRASIL

SETEMBRO DE 2022

Beltrão, Alessandro Caetano

A Low-Cost Continuous Development Architecture Based on Container Orchestration / Alessandro Caetano Beltrão. – Rio de Janeiro: UFRJ/COPPE, 2022.

XIV, 72 p.: il.; 29,7 cm.

Orientadores: Guilherme Horta Travassos

Breno Bernard Nicolau de França

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2022.

Referências Bibliográficas: p. 65 – 67.

1. Engenharia de Software. 2. Internet das Coisas. 3. Engenharia de Software Experimental. I. Travassos, Guilherme Horta *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Dedico este trabalho aos meus pais e amigos

Agradecimentos

Agradeço a minha esposa Rayane, pelo amor, carinho, compreensão e por ter me acompanhado em nos desafios da minha vida, o que inclui este mestrado. Agradeço também aos meus pais, Mario e Diva, que me apoiaram, confiaram em mim e me ajudaram em mais esta realização.

Agradeço aos meus orientadores, Guilherme Horta Travassos e Breno Bernard Nicolau de França pelos ensinamentos, por todo o apoio e paciência, pois sei que não fui um aluno fácil.

Agradeço aos professores Claudio Miceli de Farias e Lincoln Souza Rocha por aceitarem avaliar este trabalho.

Aos meus amigos de Brasília e amigos do Rio de Janeiro que me inspiraram e me incentivaram.

Agraço ao Hilmer por ter me apresentado a oportunidade de realização deste mestrado, a todas as conversas e conselhos durante estes anos.

Agradeço aos amigos do grupo de Engenharia de Software Experimental, em especial a Andréa, Bruno, Helvio, Luciana, Paulo Sérgio, Rebeca, Talita e Victor Vidigal, pelas conversas, risadas e pelo apoio nos assuntos de pesquisa e no laboratório.

Agradeço também à equipe da UFRJ, da COPPE e do PESC, pelo suporte durante todos esses anos. Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo apoio para a realização desta pesquisa.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

A Low-Cost Continuous Development Architecture Based on Container Orchestration

Alessandro Caetano Beltrão

Setembro/2022

Orientadores: Guilherme Horta Travassos

Breno Bernard Nicolau de França

Programa: Engenharia de Sistemas e Computação

Este trabalho apresenta um modelo de arquitetura para uma plataforma para desenvolvimento contínuo de sistemas baseados em Internet das Coisas (IoT). Essa arquitetura é desenvolvida com base em técnicas de virtualização e conteinerização de aplicações que devem então ser distribuídas e implantadas em dispositivos considerados restritos em relação a recursos de hardware, energia, rede e entrada/saída. A arquitetura desenvolvida neste trabalho foi implantada em um conjunto de Hardware de forma a simular um ambiente IoT e então testada e verificada experimentalmente através de simulações. Os resultados obtidos juntamente com sua utilização em projetos de sistemas de software IoT corroboram a viabilidade e usabilidade de tal arquitetura quando utilizada em um ambiente IoT.

Abstract of Dissertation Presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

A Low-Cost Continuous Development Architecture Based on Container Orchestration

Alessandro Caetano Beltrão

September/2022

Advisors: Guilherme Horta Travassos

Breno Bernard Nicolau de França

Department: Systems Engineering and Computer Science

This work presents an architectural model for a platform for the continuous development of Internet of Things (IoT) based systems. This architecture is developed based on virtualization and containerization techniques for applications that must be distributed and deployed on devices considered constrained in terms of hardware, power, network, and input/output resources. The architecture developed in this work was implemented in a set of hardware to simulate an IoT environment and then tested and verified experimentally through simulations. The results obtained and their use in IoT software systems projects corroborate the feasibility and usability of such an architecture when used in an IoT environment.

Index

Index of Figures

Index of Tables

1 Introduction

In this Chapter, we present the context and research question regarding the development of this work. After that section, we discuss the objectives and methodology that guided the execution of this dissertation, and at the end, we describe the general structure of the dissertation.

1.1. Motivation and Context

In recent years, society has been passing through a transition in how to interact with systems and technologies. Some authors (ATZORI; IERA; MORABITO, 2010) describe this new period as the Fourth Industrial Revolution, or Industry 4.0. In which applications, devices, and software systems permeate the life of people, execute tasks, control other software systems and work together to make the life of people easier. According to (MOTTA; SILVA; TRAVASSOS, 2019), these new systems, in the context of the Internet of Things (IoT), can be ubiquitous, have cyber-physical characteristics, and are context-sensitive, making them unique from other conventional systems.

IoT software systems have specific characteristics that differentiate them from conventional software. The study (MOTTA, Rebeca, SILVA, et al., 2019) characterizes IoT in a subcategory from Industry 4.0, identifying twenty-nine properties and three possible behaviors common in IoT: identification, sensing, and actuation. The identification behavior functions to identify *things and* objects from the real world*;* labeling them and enabling them to have an identity can then be recovered and broadcast to other things. The primary function of the sensing behavior is to sense the environment information, which it must aggregate and transmit; this behavior enables the IoT systems to be aware of the real world. The last possible behavior is actuation, where the device can interact with the real world using mechanical inventions based on the data received from other sensing devices or a command from the user.

Based on the characterization already presented, the IoT term was proposed and defined as a group of accessible technologies and devices that interact with a network and are capable of actuating in some context, exchanging data with other devices, and execute commands, and making decisions (MOTTA; SILVA; TRAVASSOS, 2019). In light of these characteristics, it is possible to affirm that the construction of software for the IoT context is a demanding and troublesome activity since it must consider all the specific features of the IoT domain (ZAMBONELLI, 2017).

Deployment activities are relevant for any software development project since they refer to delivering the final application package to the client. In modern software development teams, developers often choose to perform the deployment using automated tools in a continuous cycle, a process in the development process called continuous deployment. There is already a comprehensive discussion regarding continuous software deployment techniques for conventional software systems (HUMBLE; FARLEY, 2010). However, software deployment activities are still being discussed in IoT applications since there is no consensus regarding using tools and techniques.

Some studies, like those by (LWAKATARE et al., 2016; MOORE et al., 2016), discuss the difficulties of continuous deployment activities, specifically software containers to deploy applications on IoT devices. A recent study by (GEORGETA GUŞEILĂ; BRATU; MORARU, 2019) also proposes a continuous deployment pipeline for IoT; however, it does not use software container tools for packaging and delivering the software. The steps of the pipeline proposed by (GEORGETA GUŞEILĂ; BRATU; MORARU, 2019) are in order:

1. Source code checkout
2. Automated build
3. Automated packaging
4. Unit testing
5. Automate deployment to the development environment
6. Stress, security, and system testing
7. Automated deployment to the staging environment
8. Automate functional and regression testing
9. Automated monitoring

Considering all the cited studies and the characteristics of IoT applications, we propose an architecture for the continuous development of IoT software systems in this dissertation as an adaptation of the steps already shown in the literature for conventional applications. We based this architecture on modern open-source software development tools to allow the deployment of software containers in a highly available environment using Kubernetes as an orchestrator. To simulate IoT devices, we used the Raspberry Pi platform, a popular hardware choice in IoT applications, to host the cluster and integrate it with Gitlab. This code collaboration tool allows the deployment and building of applications in different environments. In the end, we used the proposed architecture

in diverse applications developed during some disciplines of the COPPE/UFRJ Software Engineering graduation course.

1.2. Problem and Research Question

The Internet of Things (IoT) paradigm has recently changed how applications, people, and devices interact (ATZORI; IERA; MORABITO, 2010). More and more devices are integrating into our personal lives and activities to accomplish this new level of interaction with the end-user. For example, they may collect data, act as hubs for communication between other devices, and work collaboratively. As IoT becomes widespread, challenges regarding security, scalability, communication and networking also surface, creating a favorable scenario for more investigation and research.

In IoT, a common challenge is the provisioning and deployment of the application on the devices. It is relatively easy to provision the devices manually in a toy application. However, in a real-world application, the number and the geographical distribution of the devices can turn this task arduous and complicated. Moreover, to perform such deployments continuously (HUMBLE; FARLEY, 2010), i.e., on a regular and frequent basis, is an even more strenuous activity in the IoT context. Therefore, some studies such as Rufino et al. (RUFINO et al., 2017), Morabito and Lwakatare et al.(LWAKATARE et al., 2016; MORABITO, 2016), and Moore et al. (MOORE et al., 2016) explore the possibility of utilizing Linux container technologies, especially Docker, to enable scalable deployments on IoT devices since Docker and containers are widely used for deployments of microservices into the cloud.

Based on the difficulties of the studies already cited and to contribute to the topic of enabling and testing software containers to scale deployments on low-cost devices, this dissertation aims to answer the following research questions:

- Can conventional software container orchestration tools enable continuous deployments on low-cost devices?
- What are the limitations of these software container orchestration tools in the limited environment provided by low-cost devices?

1.3. Objectives

This work aims to analyze the feasibility of using software containers and software container orchestration tools to deploy software systems in low-cost devices

that may be used in IoT scenarios. Specifically, we aim to understand and discuss the limitations and hardships of using technology already proposed for conventional software and deploying software in highly heterogeneous and more restricted environments, which are common to IoT hardware.

Based on this premise, this work first characterizes the devices and technologies it aims to evaluate. Then, based on familiar IoT scenarios reported in the technical literature, consider these technologies regarding response times and the number of possible connections. The specific objectives of this work are described below:

- To evaluate the performance of the software containers and container orchestrators on IoT devices.
- To compare popular container orchestration tools regarding their performance on devices that use the ARM processor architecture, specifically Raspberry PIs, general-purpose hardware widely on the market.
- To propose an architecture for the continuous development of IoT applications.

1.4. Methodology

In this research, we adapted the proposed methodology of Shull, Carver, and Travassos (2001). The evidence-based methods consist of a series of experimental studies to improve the software technology that will be conceived, constructed, and adopted. In the case of this research, the technology already exists but is not commonly used in the proposed context. Below are the steps taken to evaluate the IoT technology scenario.

- Bibliographic Review: the search for secondary studies to understand the state of the art regarding the usage of software containers on contemporary software systems.
- Technical Proposal: The architecture proposal will later be evaluated based on the bibliography.
- Evaluation of the Technology: The conception of experiments to evaluate the technology regarding the metrics (response time, number of connections) proposed in the objectives of this study.
- Comparison with Similar Technologies: The comparison of available technologies to evaluate the pros and cons of each of these technologies.

- Building of the Proposed Architecture: The formation of the proposed architecture and the usage of projects currently in construction in the Laboratory of Experimental Software Engineering (LENS ESE COPPE/UFRJ).

1.5. Contributions and Publications

Different contributions can be observed with the reading of this master's dissertation. Some of the contributions are:

- Present the reader with the current studies regarding container technologies in the context of IoT as described in technical literature.
- Performance evaluation of Kubernetes, a popular container orchestration tool, as a deployment platform for IoT Devices.
- To compare three container orchestrators used in IoT systems on ARM-based hardware.

During the realization of this master's dissertation, we had a publication regarding the performance evaluations experiments:

- Beltrão, A. C., de França, B. B. N., & Travassos, G. H. Performance Evaluation of Kubernetes as Deployment Platform for IoT Devices. In: Proceedings of the XXIII Iberoamerican Conference on Software Engineering, CIbSE 2020, Curitiba, Paraná, Brazil, November 9-13, 2020

1.6. Work Structure

We organized this master's dissertation into another five chapters, besides this first introductory one that describes the context, motivation, and problems to which this work is related. The textual structure of this work follows the following format:

Chapter 2 – Bibliographic Review: Describes the review to search related concepts to the Internet of Things and Container Technologies used in this domain.

Chapter 3 – Performance Evaluation of Kubernetes on IoT Devices: Presents the tests and results of a performance evaluation of Kubernetes, a container orchestration tool, on devices commonly used on IoT systems.

Chapter 4 – Performance Comparison of Container Orchestration Tools on ARM Platforms: This Chapter evaluates three of the most prominent container orchestration tools: Kubernetes, k3s, and KubeEdge, in the context of devices that use the ARM processor architecture.

Chapter 5 – Architectural Proposal for Continuous Development on IoT Devices: In this Chapter, we present our architectural proposal for continuous deployment of IoT devices and offer the data on how it was used in the last two years from software development groups on Coppe/UFRJ.

Chapter 6 – Conclusion: Describes the conclusions and contributions from this work; it also presents the future outcomes and new perspectives for this research.

2 Bibliographic Review

This Chapter aims to familiarize the reader with relevant concepts regarding the technologies explored in this dissertation. First, we scour the Internet of Things concepts and their challenges. Then, we explain the container technologies and how new implementations of such techniques may be adopted to implement continuous delivery on contemporary software systems.

2.1. Internet of Things

The Internet of Things paradigm came to attention in 1999 in a speech from Kevin Ashton to the Procter & Gamble company. Since then, this new model has started gaining more attention, gaining ground in wireless communications and automation scenarios. The basic idea behind IoT is that pervasive hardware and software, or *things*, can communicate and interact with each other to accomplish specific tasks. In this context, domestic activities such as assisted living, enhanced learning, and e-health are scenarios in which this paradigm can lead in the future. However, there are still extensive computation and telecommunication challenges to address to allow the complete interoperability of devices as envisioned by IoT.

As the Internet of Things permeates many knowledge areas, this paradigm has many definitions. Some authors (KATASONOV et al., 2008; SZILAGYI; WIRA, 2016; TOMA; SIMPERL; HENCH, 2009) approach the semantic aspects of IoT and the challenges regarding the extremely high demand for the organization, storage, and search of data produced by the devices. The study (TOMA; SIMPERL; HENCH, 2009) further explores scenarios regarding semantic execution environments, data modeling, and the increasing storage needs

Later, the *survey* (ATZORI; IERA; MORABITO, 2010) aimed to familiarize the reader with the manifold meanings given to this paradigm based on the literature. According to this survey, the first devices, or things, that represent IoT is Radio-frequency Identification Tags (RFID). It composes a very "Things-Oriented View" of IoT, as such devices and technologies appear in IoT-related studies as enabling technologies (DORSEMAINE et al., 2015; RAZZAQUE et al., 2016; XINGMEI; JING; HE, 2013). However, such technology does not represent the full extent of IoT.

Apart from the "Things-Oriented View," two other perspectives for IoT are also presented by (ATZORI; IERA; MORABITO, 2010), an "Internet-Oriented" and "Semantic-Oriented," representing networking concerns such as unique addressing and security and the representation and storage of information, respectively.

Regarding the Internet-Oriented perspective, different works published in the literature aim to tackle problems related to the large scale of the network infrastructure needed for IoT scenarios. For example, the heterogeneous aspect of IoT devices and network protocols used for communication, like *Bluetooth*, *NFC*, *Zigbee*, *Sigfox*, *LoRa*, *NB-IoT,* and others as well the numerous challenges regarding the addressability of the devices, data transfer, and security concerns, are explored in studies by (BORMANN; ERSUE; KERÄNEN, 2014) and the subsequent ones by (C. KOLIAS, G. KAMBOURAKIS, *et al.*, 2017) and (KAWABATA, ISHIBASHI, *et al.*, 2017).

Studies like the one by (KAWABATA et al., 2017) propose new network models and architectures to enable large-scale IoT. While other studies, such as the one by (MEKKI et al., 2019), compare network protocols in terms of energy consumption, coverage, quality of service (QoS), scalability, deployment, and others. Studies conducted by (BREWSTER et al., 2017) address the lack of a unified security vision for IoT. (C. KOLIAS et al., 2017) discuss the use of *Mirai Botnets,* a malware used for DDoS attacks in IoT scenarios, where a bot can invade a device and use it as an attacker to another service.

Although the three views from the Internet of Things (ATZORI; IERA; MORABITO, 2010) represent the *middleware* (Internet Oriented), the *things* (Things Oriented), and the *knowledge* (Semantic Oriented), define the concept of the Internet of Things as a multidisciplinary paradigm.

In more recent works, (MOTTA; SILVA; TRAVASSOS, 2019) conducted a secondary study to characterize the properties of IoT-based systems. It identified twenty-nine characteristics: accuracy, adaptability, availability, connectivity, efficiency, extensibility, flexibility, manageability, modularity, performance, privacy, and others, and the three possible behaviors in IoT applications: acting, identification, and sensing.

Based on the identified characteristics, (MOTTA; DE OLIVEIRA; TRAVASSOS, 2019) proposed seven facets to characterize the multidisciplinary IoT systems: connectivity, things, behavior, smartness, problem domain, interactivity, and environment. In that regard, the facets presented by Motta et al. (2019) corroborate and

problems and challenges presented by authors like Atzori et al. (2010) but expand them by defining mapping concerns to each different facet.

2.2. Container Technologies

Software containers are sandboxed environments containing all the tools and elements for software to run in any environment. Virtualization technologies, containers, and virtual machines (VMs) have similar objectives of isolating processes and running them in a controlled environment to preserve the hosting environment. However, they differ in architecture and execution. A virtual machine emulates every aspect of a real computer, including its hardware and software components. A *Hypervisor* is necessary to create a virtual machine. The *hypervisor* is the software responsible for handling *system calls* and executing instructions on the virtualized hardware. There are two classifications of *Hypervisors: type 1* or *bare-metal,* in which the *hypervisor* communicates directly with the physical hardware components of the machine, and *type 2* or *hosted*, where another operating system hosts the *hypervisor* (MERKEL, [s.d.]).

However, unlike virtual machines, the containers are based on a group of system calls embedded into two components of Linux, the *cgroups* and the *namespaces. Cgroup*, or control group, is a feature from the Linux Kernel that "allows the organization of processes into hierarchical groups whose usage of various types of resources can then be limited" (cgroups(7) - Linux manual page, [S.d.]). The implementation of *cgroups* was a contribution from Google that aimed to create a single interface to manage processes in a Linux environment. Currently, *cgroups* are capable of the following:

- **Resource Limitation**: Resource groups are needed to allow users to assign memory, CPU time, and disk space to a process group. That group should never exceed these limits.
- **Prioritization:** This allows the assignment of different priorities to processes on the system scheduler.
- **Accounting:** Its users can monitor the resources used by a control group.
- **Control:** Users can decide to freeze, stop, or kill a process in the control group.

The merge of *cgroups* into the Linux Kernel allowed the development of new technologies, such as the *Linux Containers (LXC). This interface* handled the *system call* on *cgroups* and made creating containers easier. LXC also implemented the concept of *namespaces,* a process visibility isolation technology, allowing the creation of a sandbox environment based on a minimal Linux installation and decoupling of software pieces

(namespaces(7) - Linux manual page, [S.d.]). The Linux Kernel is compatible with six different namespaces they are mount, unique time-sharing (UTS), interprocess communication (IPC), network, process id (PID), and user namespace. All these new features in the Linux kernel allowed the creation of fresh and more advanced container technologies.

2.3. Docker

Docker is a container creation and management technology that used LXC as the backend for container creation during its first year. However, after the contributions from a Google team with the LMCTFY, Docker started using *libcontainer* as the backend. Independently from the library used for container creation, *libcontainer,* or LXC, Docker still uses *cgroups* and *namespaces* on a Kernel level (Docker Documentation | Docker Documentation, [s.d.]). Figure 2 shows a comparison between Docker and conventional virtualization methods.

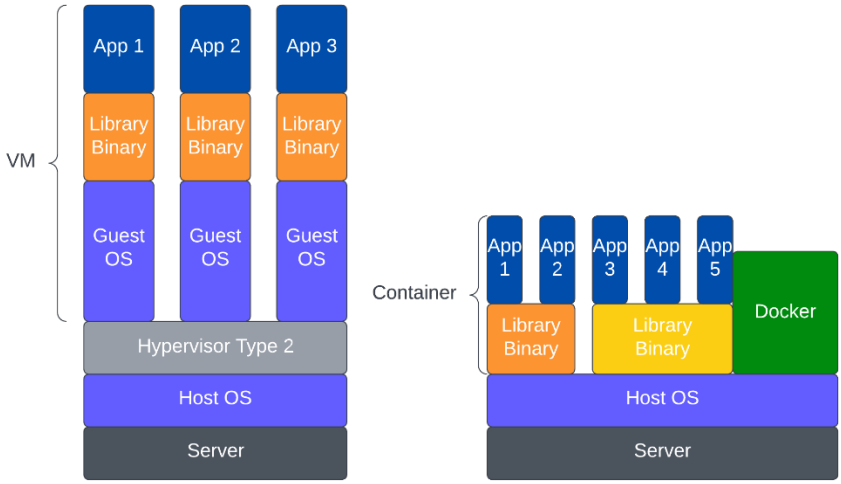

Figure 1 - Comparison of Docker and Conventional Virtualization Methods (Adapted from ("Docker Documentation | Docker Documentation," [S.d.]))

Today, the distribution of Docker happens through the Docker Engine, one application composed of three components, a *daemon*, implemented as a *systemd* service called *dockerd*, a command-line tool, and an API. A typical use case for Docker is that a client communicates with the API through the command-line interface (CLI). The API communicates with the *daemon,* responsible for creating the container based on an *image. An image* is a frozen set of alterations on a filesystem. An essential addition to Docker is the Container Registry, an application allowing users to store *images* for later use. Figure 3 represents the Docker architecture and how all the services interact.

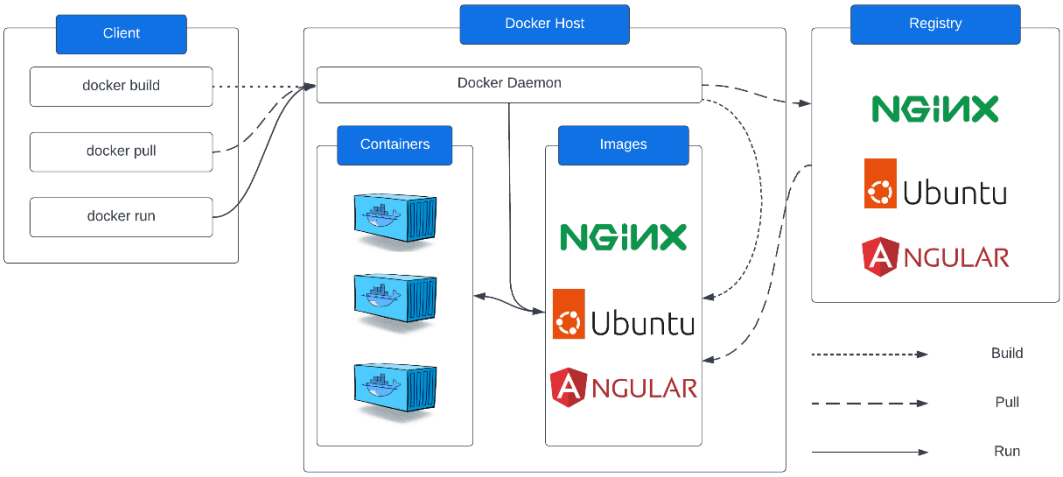

Figure 2 - Docker Architecture (Adapted from ("Docker Documentation | Docker Documentation," [S.d.]))

Docker and LXC are fundamentally similar, as they both serve the same purpose: manage *userspaces* and isolate resources. However, these two solutions implement different approaches regarding process and state management and portability, as described in the following.

- **Process Management**: All containers in Docker run as a single process. This way, an application consisting of *n* concurrent processes also need *n* containers to run, each handling one unique operation. On the other side, in LXC, each container has an *init* process that allows the execution of many processes.
- **State Management**: Docker is stateless, not allowing persistent storage. After creating an image, it consists of a group of read-only layers. So, when this image starts executing, a container is made. Every time the container is modified, it compares its current and initial state. This comparison, called *diff*, is kept by Docker until the creation of a new image or the container termination when Docker discards all the changes on the *diff*. Fig. 1 shows an example of that behavior.
- **Portability**: Unlike LXC, Docker is much more efficient in abstracting the network layer, storage layer, and operating systems from applications and containers. These abstractions allow developers to deploy applications into different environments transparently, as no further configuration is needed relating to the target operating system where the container application will run.

2.4. Container Technologies in IoT

Container technologies provide an alternate option for software developers to create an encapsulated sandbox environment to run their applications. This sandbox environment can be later uploaded to a container registry (a repository for container images), allowing other developers to pull these images and execute them on demand. Using such technology in the Internet of Things scenarios is a recurrent research topic, as some authors have already explored the possibilities of using containers to deploy an application in IoT devices.

A study was already conducted by (RUFINO et al., 2017) on using Docker as the driver for deploying microservices in IoT. In this study, they propose a three-layered architecture. One layer represents the Cyber-Physical System (sensors and actuators), another represents the intermediary gateways, and finally, in the cloud, the enterprise systems. In the proposed architecture, it is defined by design that data should travel from the constrained nodes to the more capable ones. At the same time, the cloud is responsible for containing the *business rules* and control of the less capable nodes.

Morabito et al. published three studies in 2016 and 2017 regarding the usage of containers in IoT. The first study executed a synthetic benchmark of Docker in *Raspberry Pis* regarding hardware resources, like CPU, memory, disk I/O, and network I/O. In the end, the tests show that Docker had a negligible impact on the observed parameters compared to a native execution (MORABITO, 2016). In their second study, they executed a performance test comparing models from boards from two different vendors, *Raspberry* and *ODROID* (MORABITO, 2017). *S*pecifically, they compared *Raspberry Pi 2* and *3 Model B* and *ODROID C1+, C2,* and *XU4.* Then, using *Sysbench,* they observed the usage of hardware resources on both platforms. In all cases, Docker had an impact of approximately 5% in terms of performance compared to native executions, a result that kept valid even after running several instances of the virtualized application. Also, in the end, the ODROID boards, notably C2, outperformed all the other models. Finally, in their third work, they compared two possible container-based approaches for IoT service provisioning, *Container-based Pair-oriented IoT Service Provisioning* (CPIS) and *Container-based Edge-Managed Clustering* (CEMC) (MORABITO et al., 2017). In the CEMC approach, a container orchestrator manager is responsible for deploying and managing services in *Container-based IoT Workers* (CIW). In contrast, in a CPIS approach, there is no manager or container orchestrator as an intermediary for the requests, and the devices must cooperate in allocating the containers. Although in all

works published by *Morabito et al.,* the comparison was the usage of hardware resources between Docker and bare-metal applications, they did not explore the use of container orchestrators as in the presented dissertation.

Another study tackling the usage of containers in the context of an IoT application is the one performed by (MOORE et al., 2016). They proposed a solution based on Docker and the Elasticsearch, Logstash, and Kibana (ELK) stack. Their goal was to collect logs from the use of solar energy in the households of Milton Keynes, a city in England. The study showed promising results as a real-case scenario for Docker in the context of Smart Cities.

2.5. Kubernetes

Kubernetes (k8s) is an open-source platform that manages services and workloads based on containers. Google created Kubernetes to be portable and extensible, using a declarative language to allow easy configuration and automation. By concept, Kubernetes is designed for cloud computing and easily deployed on different cloud providers, such as Google Cloud Platform (GCP), Microsoft Azure, and Amazon Web Services. It is also possible to deploy Kubernetes in bare-metal clusters. However, native features like Layers 4 and 7 load-balancing are not supported.

As discussed in the previous sections, using containers to package and run applications can be very beneficial. However, in a production environment, there may be a need for more control and orchestration, and k8s is a way to address that problem. Among the features that k8s include, we mention some of the most relevant:

- **Load balancing and service discovery**: Kubernetes can expose a container through a DNS or IP address. It can also detect and control network flow to keep applications healthy.
- **Storage Orchestration**: Kubernetes allows the users to store data by creating system mount points on local, public, or cloud networks.
- **Automatic Rollouts and Rollbacks**: Users can describe a designated state for Kubernetes containers, and it will make the necessary changes so that containers in a current state can migrate to the new condition in a controlled manner. This way, users can, for example, create containers for deployment, remove old containers, and migrate all resources to the new deployment.
- **Self-healing**: Kubernetes can restart and remove failed containers.

Because Kubernetes works in a master-slave architecture, it is necessary to delineate all components composing the cluster and which nodes (master or slave) each component resides. Besides, we present essential concepts related to Controllers and the Basic Objects that make up the Kubernetes API based on the Kubernetes Documentation ("Kubernetes," [S.d.]).

2.5.1. Master Node Components

In k8s, the primary node is the cluster control pane. It must make decisions about the cluster, such as resource management and workload scheduling, as well as responding to events. The components of primary nodes are:

- **Kube Apiserver**: the component responsible for exposing the Kubernetes API. It is considered the Kubernetes front-end.
- **Etcd**: it is key-value-based storage used by Kubernetes to store the cluster meta-data.
- **Kube Scheduler**: a component that observes the creation of new pods not yet assigned to a node and selects one so the pods can run.
- **Kube Controller Manager**: the component responsible for running cluster controllers. The controllers are:
    - **Node Controller**: the controller responsible for detecting and responding to the disconnection of a node from the cluster.
    - **Replication Controller**: the controller responsible for maintaining the correct number of pods for each replication object.
    - **Endpoints Controller**: it is responsible for joining Kubernetes services and pods.
    - **Service & Token Controllers**: create default system accounts and generate new API keys for namespaces.
- **Cloud Controller Manager**: Component responsible for communicating with cloud providers such as Amazon Web Services (AWS) and Microsoft Azure.

2.5.2. Default Node Components

General Kubernetes components run on all nodes, keep pods running, and provide the Kubernetes runtime environment. These components are:

- **Container Runtime Environment:** software responsible for creating and executing containers can be *Docker*, *d-container*, *create-it*, *rktlet*, or any other Kubernetes compatible Container Runtime Interface (CRI) implementation.
- **Kubelet**: Kubernetes agent that runs on all cluster nodes and serves to execute the containers in the context of a pod.
- **Kube Proxy**: a proxy that runs on all cluster nodes and implements the Kubernetes service layer.

2.5.3. Objects

Kubernetes has a set of abstractions to represent the state of the system, deployed applications, containers, workloads, hardware attributes, and other information about the current state of the cluster, *objects* in the Kubernetes API, represent these abstractions. The primary Kubernetes objects are:

- **Pod**: Pod is the basic unit of execution for a Kubernetes application, the most straightforward unit of the Kubernetes object model. It encapsulates the container, or an application's containers, with a unique network IP, storage resources, and all the necessary configurations to define how the container should perform.
- **Service**: an abstract way to expose an application running on a pod as a network service. Since pods are transient, they can automatically die and resurrect, and since each pod has a unique IP, a set of pods may not have the same IPs sometime after creation. Therefore, it is essential to define a service to abstract the IPs of pods and maintain a fixed IP in addition to establishing access policies.
- **Volume**: the files on the container disk are also temporary, which is a problem for nontrivial applications. On the destruction of a container, the *kubelet* restarts it, but it will start in a clean state, so all previously present files are lost. To get around this, Kubernetes allows the user to create volumes or directories on disk inside the container, which will not be lost when the container is destroyed.
- **Namespace**: Kubernetes supports multiple virtual clusters hosted on the same physical cluster. These virtual clusters are called namespaces.

2.5.4. Controllers

Controllers are control loops that observe the state of the cluster and make changes as needed. Conceptually the cluster has a desired state and a current state, so

each controller works to make the current state as close as possible to the desired state. The types of controllers are:

- **Deployment**: it provides declarative updates of pods and replica sets. The user declares a required state, and the controllers themselves are responsible for deployments by changing the current state of the cluster or the new state in a controlled manner.
- **DaemonSet**: it ensures that at least one instance of a given Pod runs on each node. This kind of controller is useful for solutions that require an *agent* running on each node, as it can ensure that the agent will always be available. Everyday use cases for *DaemonSets* are logging services such as *logstash* and *fluentd* and storage services such as *ceph* and *glusterd*
- **StatefulSets**: it ensures pod ordering and uniqueness. Each pod in a StatefulSet is not interchangeable and has a persistent identifier that remains even after rescheduling.
- **ReplicaSets**: it aims to maintain a stable number of pod replicas and is usually used for high availability.
- **Jobs**: a job creates one or more pods and ensures that several will have minimal success. Usually is used for one-shot scripts and services.

2.6. Other Solutions on the Market

In this section, we explore the available solutions on the market that support overcoming issues common to IoT. For each solution, the main objectives and limitations are delineated.

2.6.1. K3S

K3S is a Lightweight Kubernetes distribution optimized to operate in constrained devices and designed to work with IoT devices. K3S substitutes Kubernetes Engine, a fully compliant Kubernetes distribution distributed as a single binary. K3S achieves a lower impact on performance by removing some of the plugins that are present in the conventional Kubernetes installation, such as the support for cloud providers (Amazon Web Services, Azure, Google Cloud Platform) and the storage plugins, as the GCE PersistentDisk, AWS ElasticBlockStorage, Azure Disk, Storage OS and others. It also substitutes *Etcd,* in-memory key-value storage used as the conventional datastore for k8s, with SQLite, a file-based database; it has a more straightforward installer that

handles most of the overhead of Kubernetes installation and has minimal dependencies, such as *containerd, Flannel, CoreDNS, CNI* and *iptables,* and *socat.* However, by default, K3S does not support high availability environments, as it uses SQLite, but it is possible to configure external databases to achieve high availability ("K3s - 5 less than K8s", [S.d.]).

2.6.2. KubeEdge

KubeEdge is an open-source system that extends Kubernetes capabilities to work with Edge Devices. According to its documentation ("What is KubeEdge — KubeEdge Documentation 0.1 documentation", [S.d.]), it is supposed to provide infrastructure support for networking, application deployment, and metadata sync between edge devices. However, unlike K3S, KubeEdge does not replace Kubernetes, as it requires a Kubernetes primary node to work, replacing only the worker nodes. KubeEdge aims to provide lightweight abstractions, so hardware with less capacity can run as worker nodes. The main components of KubeEdge are:

- **Cloud Core**:
    - **CloudHub**: a WebSocket responsible for caching and sending messages to the edge nodes.
    - **EdgeController:** an extension responsible for integrating the existing Kubernetes API Server and the CloudHub.
    - **DeviceController:** the component responsible for managing devices in KubeEdge. It describes device metadata and status and synchronizes these devices between the cloud and the edge.
- **Edge-Core:**
    - **Edged**: agent that runs on each node and manages containerized applications. Act as a substitute for *kubelet* to create and manage pods.
    - **EdgeHub**: A WebSocket responsible for interacting with the Cloud Service. It substitutes the API calls from the *kubelet* to the Kube API Server using this WebSocket through HTTP.
    - **EventBus:** KubeEdge has a built-in MQTT client that interacts with MQTT servers. On the documentation, KubeEdge recommends the usage of Mosquitto as an MQTT Server.

- **DeviceTwin:** Component responsible for storing the device status and syncing with the cloud. It is also responsible for registering new devices added to the cloud or edge. The main goal of DeviceTwin is to create an asynchronous way to store the meta-data from Kubernetes.
- **MetaManager:** Responsible for intermediating messages between Edged and EdgeHub, it also provides an interface for storing data in the DataStore.
- **DataStore:** Substitutes the default *Etcd* datastore from Kubernetes with an SQLite instance. SQLite is a c-language library that implements a small and fast SQL database engine.

One of the core features of KubeEdge is the built-in MQTT client's use of a lightweight MQTT broker to communicate between devices, applications, and *DeviceTwin.* KubeEdge automatically creates topics in the broker to get details from members, update the states, or perform twin operations on devices.

Since KubeEdge is not a full-fledged Kubernetes installation, it is possible to use K3S and KubeEdge together to create a Kubernetes cluster if the cluster configuration does not allow a primary node in the cloud.

2.6.2.1. KubeEdge Limitations

KubeEdge provides some advantages regarding the synchronization of devices. However, it also has some drawbacks compared to a default Kubernetes node. Currently, the KubeEdge node needs to be on the same subnet, as it is incompatible with the Kube Proxy to communicate with the worker nodes. As a result, pod-to-pod communication can only happen on the same subnet. Furthermore, KubeEdge does not support the usage of Layer 7 load balancers and service discovery as it is not compatible with any container network interface.

KubeEdge also does not support high availability; users cannot deploy the *CloudCore*, the cloud component, on multiple instances. In addition, KubeEdge does not have monitoring capabilities for the edge nodes and does not support other protocols like AMQP, Zigbee, and Bluetooth.

2.7. Chapter Considerations

This Chapter presented the two main concepts related to this work, the Internet of Things paradigm and its definitions and the use of containers for software deployments on IoT devices.

The first topic aims to familiarize the reader with a general understanding of IoT and how it is perceived by modern literature from a generalist view. Then, we present how containers work and their importance as a modern software development tool to allow deployments in different environments. Furthermore, such technology creates high-availability environments while achieving security and performance requirements.

Regarding the use of containers in IoT, the literature shows that there are currently efforts to validate such technologies. However, none of the studies presented here compare available technologies for containers and orchestrators in the IoT context, as the studies compare the performance of bare-metal applications and Docker in terms of hardware resource consumption, like CPU, RAM, and energy. Furthermore, the presented literature does not compare Kubernetes and other IoT-focused container orchestration tools in actual or simulated scenarios. Nevertheless, the current efforts by software companies to modify Kubernetes to this context show that the usage of such technologies could be promising for the future.

3 Performance Evaluation of Kubernetes on IoT Devices

In this Chapter, we present a performance test of Kubernetes, a popular container orchestration tool on the Raspberry PI platform. We created two common scenarios to conduct the tests: a synchronous HTTP protocol and an asynchronous one using the AQMP protocol.

3.1. Introduction

The adoption of Kubernetes as a driving technology for IoT is still under research. This way, we understand the need to evaluate such technology combinations.

The first requirement to use Kubernetes in a real scenario is the availability of hardware capable of operating as Worker and Primary nodes. However, common scenarios of IoT present less capable nodes located in the front-end, and intermediary devices on the near end can act as hubs, controlling and aggregating data before sending it to the cloud with full capabilities. Considering this, we investigated two scenarios, a synchronous one using the HTTP protocol and another asynchronous one using the AMQP protocol.

To evaluate these scenarios, we provisioned a four-node cluster using *Raspberry PIs* as worker nodes. The primary node was a workstation with four cores and eight threads x64 processors with 16 gigabytes of DDR4 *RAM*, while all the worker nodes were *Raspberry PIs Model 3B+.* We chose this setting for all scenarios, as the Raspberries are not capable of handling all the components of the Kubernetes Primary Node. Both scenarios had two main objectives, to evaluate the performance of Kubernetes on an ARM platform and to determine the limits and difficulties of using this technology on constrained devices.

3.2. Compatibilities of ARM Architecture with Kubernetes Binaries

The first impediment to using Kubernetes as a driving technology for ARM devices on IoT is the current incompatibility with the Yu (*armel*) microarchitecture for ARM processors, which makes it incompatible with processors such as ARMv6l, which

are the ones in the first versions of *Raspberry Pi*. As a result, the official repository for Kubernetes dropped the support for such architecture in 2016 ("Multi-architecture plan for Kubernetes · Issue #38067 · Kubernetes/Kubernetes", [S.d.]). Some of the reasoning behind the decision was the considerable improvements in performance on newer versions of ARM and *ppc64le* architectures while also diminishing the burden of maintaining and building binaries for older architectures. However, it is still possible to use Kubernetes on earlier ARM versions by manually compiling the code and setting up the cluster. The process is long and, in some cases, can still be used for testing purposes, yet the newer versions of Kubernetes can be too demanding for architectures older than ARMv6l.

The second limitation of Kubernetes on a constrained device is the heavy usage of *Cgroups Syscalls* to limit the resources of an application. In Kubernetes, the user may restrict the resources of an application by creating these limits in a *pod specification* file, like the one shown in Figure 4. It accomplishes these limits by using *cgroups cpuset* and *memory controllers* ("cgroups(7) - Linux manual page", [S.d.]). Nevertheless, these controllers, specially *cpuset,* may not be present on a single-core processor system, limiting the Kubernetes usage to the creation and management of containers and not being capable of creating resource limits on the pods.

```
resources:
  limits:
    memory: "128Mi"
    cpu: "500m"
```

Figure 3 - Kubernetes Resource Limits for a Pod

3.3. Experimental Setup and Procedures

In the trial setup, the first step was to disable the *Docker network bridge*, a component responsible for mapping the ports inside containers to the host network interface. The usage of the *Docker bridge* could represent another confounding factor in the trials, as the study conducted by (FELTER et al., 2015) concluded that using the *bridge* to map ports means a loss in performance regarding Network IO, while *Docker* without those configurations has the approximate performance of bare-metal applications. To allow the nodes in the cluster to communicate without interference and package loss from the firewall on the private network, we set up the nodes in a Demilitarized Zone (DMZ).

According to the Kubernetes documentation, we installed the Kubernetes version 1.17 binaries on the *Raspberry PIs*, setting them up as worker nodes. We proceeded with the installation using *Kubeadmin*, one of the provisioners of Kubernetes, while making the necessary changes to allow the execution on the *ARM* devices. We implemented the trials for both scenarios using Apache JMeter, a tool for load testing and measuring the performance of applications and servers. Moreover, during the teardown of each trial, the deployments of the applications under trial were destroyed and recreated. To provide an easy way to play the cluster and execute the trials, we implemented a handful of scripts using Chef, an configuration management tool. Using the script, we could create and tear down the cluster, keeping a reproducible initial state across all the trials.

In the synchronous scenario, we use the Alpine Linux version of the official NGINX version 1.17 Docker Image from *DockerHub*, a widely used application server and reverse proxy. This scenario represents synchronous communication between the IoT Devices (like sensors and actuators) with the IoT Gateway (deployed as Kubernetes Workers), implemented as a simple HTTP request that should return the response code 200. In this scenario, the Raspberry Pi worker node runs a pod with one container that should be able to respond to the requests from the client. Each request sent has 114 bytes in data and should get as a response a web page with 850 bytes, 238 for the header, and 612 for the body. Figure 5 presents the cluster topology for the synchronous scenario.

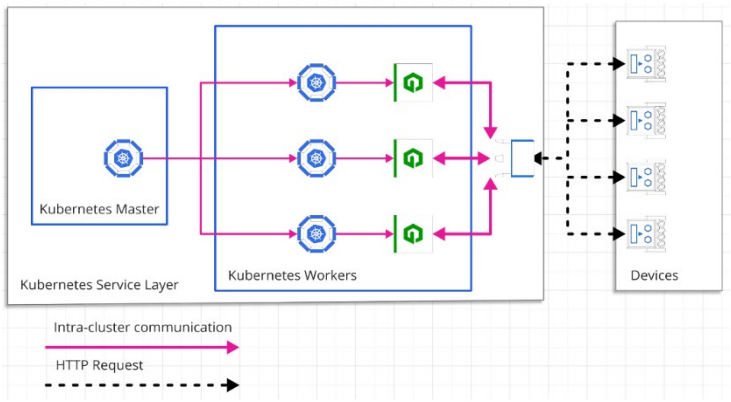

Figure 4 - Cluster Topology for the Synchronous Scenario

To evaluate the first scenario, we designed nine test cases establishing the maximum response times of the application being 100ms, 1000ms, and 10000ms based on the heuristics in (NIELSEN, 1993). These times represent the thresholds for a user's

attention span in a web application, being 100ms perceived as real-time, 1000ms as the limit for a user flow of thought, and 10000ms the limit for keeping the user's attention. We also established the number of requests per second (RPS) in powers of ten, as the goal of the performance test was to stress the cluster to a maximum, so the RPS values used are 10^1, $10^{2,}$ and 10^3. These requests are applied to the system, not just to each node. Using the defined values for the max response time and RPS, we applied them to Equation 1 ("Documentation :: JMeter-Plugins.org," [S.d.]) to calculate the total number of threads needed for each trial. Table 1 describes each one of the scenarios.

$$\frac{RPS \times Max\ Response\ Time}{1000} = Thread\ Number \quad (1)$$

Table 1 - Test Execution Parameters for HTTP Protocol Scenario

Test ID	RPS	Max Response Time (*ms*)	Thread Number
0	10	100	1
1	10	1000	10
2	10	10000	100
3	100	100	10
4	100	1000	100
5	100	10000	1000
6	1000	100	100
7	1000	1000	1000

The second scenario aimed to represent an asynchronous communication using *RabbitMQ* version 3.7.15 as a broker, and the AMQP protocol, in an IoT application using a publisher-subscriber architecture. This way, we could simulate an architecture in which many publishers send data to the gateway, and a smaller number of consumers took that data to perform another action. This scenario agrees with the settings described (ATZORI; IERA; MORABITO, 2010). Figure 6 presents the cluster configuration for the asynchronous scenario.

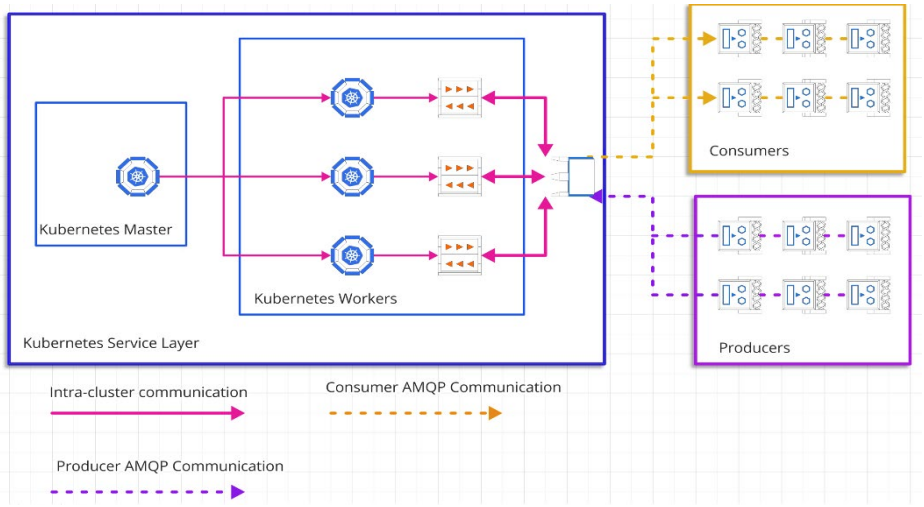

Figure 5 - Cluster Topology for the Asynchronous Scenario

We set up two trial configurations using JMeter and Apache AMQP Client (Table 2). For each trial, the publishers must deliver a message that has a hexadecimal body of 850 bytes. The delivery of the message is only considered a success when an ACK signal is received. On RabbitMQ, we configured the exchange to broadcast all messages to queues, as each trial is composed of only one queue.

Table 2 - Test Execution Parameters for the AMQP Protocol Scenario

Test ID	Subscribers	Publishers	RPS Limit on Publishers
0	1	10	100
1	10	100	1000

Similarly, as in the *HTTP* scenario, we also scaled the subscribers and publishers by powers of ten from one trial to another. Typically, in an IoT scenario, the number of publishers tends to be higher than the number of subscribers, as the number of devices on the *Front-End* (sensors and actuators) is generally higher than the number of devices in the *Near-End*. In a real publisher-subscriber scenario, each publisher sends messages repeatedly. To simulate that behavior, we established a limit on the number of requests per second the publishers can make. In both scenarios, the cluster must support the load of the trials for three minutes and twenty seconds, in which the ten starting seconds are ramp-up, and the last ten are ramp-down time. We consider that the time given for each trial scenario was enough to show the shortcomings of the architecture and the

Raspberry Pi 3B+, as the number of requests on each trial is high and already capable of stressing the cluster in some scenarios.

3.4. Results

After setting up the cluster and creating the test execution scripts using *JMeter*, we executed the scenarios. In the HTTP scenario, the cluster could handle all the requests from the first five trials with a low standard deviation from the mean response time (Table 3). Experiments 5 and 6 show that the standard deviation for the response time started to grow exponentially along with the number of threads. For a sample to be considered a failure, the server had to respond with any other status code than 200, and the request-response time should be higher than the max response time defined in the trial execution plan. Table 3 shows the test results for all the HTTP scenarios.

Table 3 - Test Execution Results from the HTTP Scenario

Test ID	Samples	Mean (*ms*)	Min (*ms*)	Max (*ms*)	SD (*ms*)	Error %	Throughput (transactions/sec)
0	1824	26	13	188	12,93	0,66%	9,6
1	1663	34	14	364	25,23	0,00%	8,3
2	1387	33	13	265	20,43	0,00%	6,9
3	18926	25	12	230	19,60	1,87%	95,1
4	18873	25	11	203	19,03	0,00%	94,4
5	17715	32	12	1028	41,08	0,00%	88,9
6	53701	350	8	21025	785,43	63,30%	263,0
7	54242	3496	8	70138	4359,41	83,42%	253,0
8	78340	23887	0	198230	37907,06	51,80%	371,8

Table 3 shows that the cluster operated well until the number of threads reached close to one hundred and the number of requests per second was close to one thousand. Then, the pod could not promptly serve all the static files as *NGINX* started creating new threads to answer the requests. Some threads needed to wait until they could be processed, creating a queue that generated high response times. In Trials 6, 7, and 8,

the major problem was the limited number of resources from the *Raspberry PI* Worker node to process all the requests simultaneously. We can observe this phenomenon by looking at the maximum throughput generated and the *RPM* expected from the trials. Figure 7 and Figure 8 present the degradation of the performance of the *Raspberries* as the number of requests became too high.

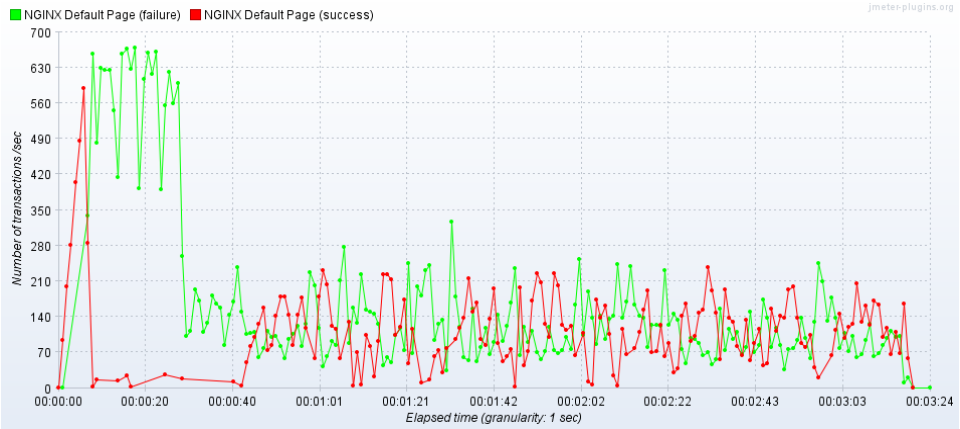

Figure 6 - Transactions for Test 7 on the Synchronous Scenario

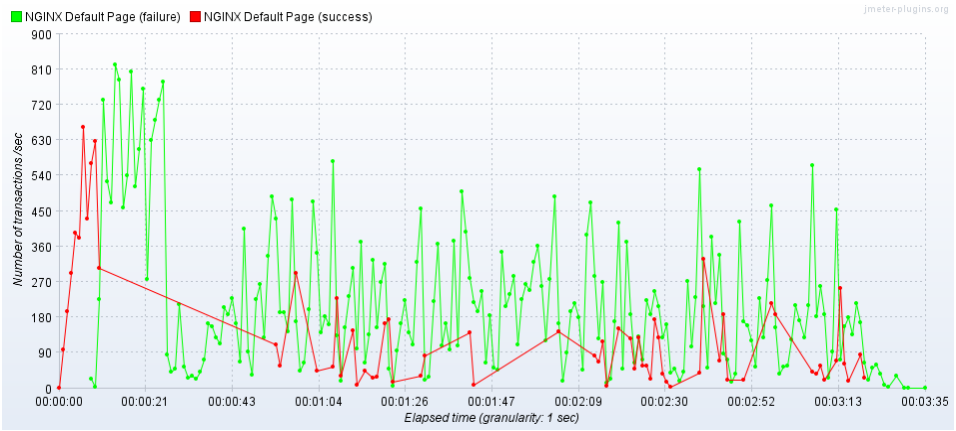

Figure 7 - Transactions for the Test 8 on the Synchronous Scenario

Regarding the trial results of the synchronous scenarios, the results show that the maximum throughput that *NGINX* from one pod with only one replica is 371,8/sec. The *Kubernetes* documentation points out ways to achieve higher throughput by horizontally escalating the architecture and allowing more replicas to coexist. Another way to improve the performance of the pods is fine-tuning the resources allocated per pod, as *Kubernetes* enables users to control the number of allocated *CPU shares* and

memory. Monitoring tools and Kubernetes may also help detect eviction signals sent by Kubernetes controllers on the *Worker Nodes*.

In the second scenario, a more lightweight protocol like *AMQP* greatly favors the *Raspberry PI* Worker nodes. In both executed trials, the results were significantly better. The first trial shows that a *Raspberry PI*, an IoT Gateway, can handle ten other devices publishing messages at a rate of 100 *m/s*. Also, it could keep the broker healthy without any failure, as it is possible to see in Figure 9.

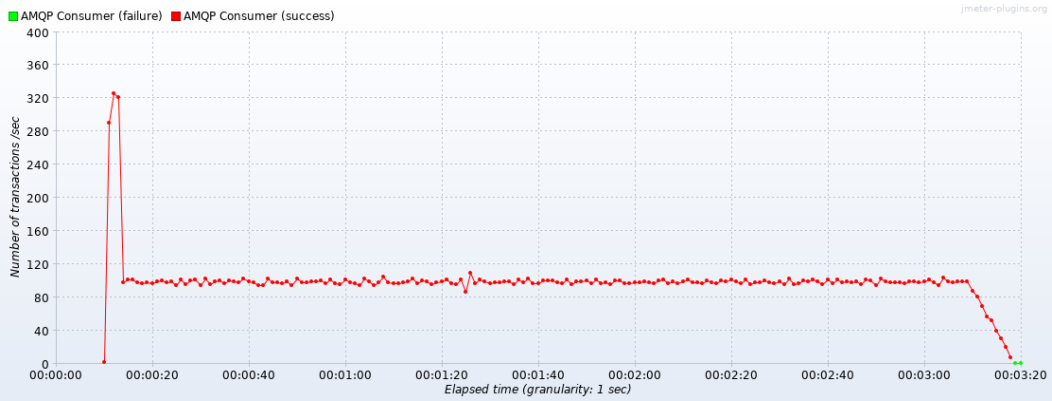

Figure 8 - Subscriber Transactions for Test 0 on the AMQP Scenario

On the second trial of the publisher-subscriber scenario, the limits of the *Raspberry PI* as a platform are more apparent. RabbitMQ reported that the cluster reached the Memory High Watermark during the trial execution. This configuration starts to block publishers' queues until the messages are clear and the memory usage lowers. The flag Memory High Watermark on the cluster is 40% of the node's total memory. We can observe these effects on the curve generated by the publisher's transactions by analyzing the dips in Figure 10.

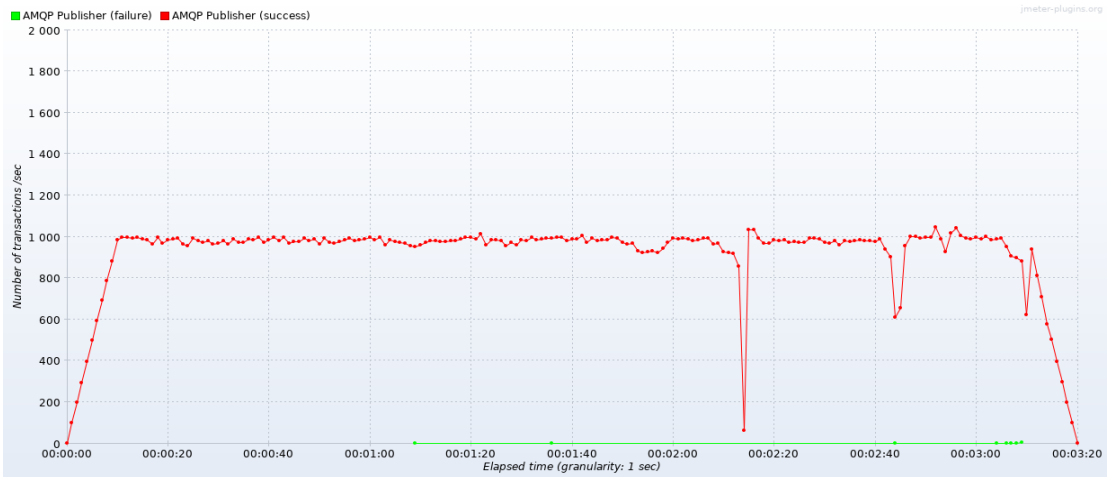

Figure 9 - Publisher Transactions for Test 1 on the AMQP Scenario

However, even activating the memory limits, the number of simultaneous publishers' connections on a pod got too high. The large connection pool ended up causing *RabbitMQ* to stop broadcasting messages to subscribers, as shown in Figure 11. The graph shows that *RabbitMQ* prioritized publisher threads, as they were in a proportion of 10:1 from subscribers. The high peaks indicate that when a subscriber thread could finally receive information, it consumed many messages simultaneously and stopped to query for new messages. At the end of this trial, the subscribers left 118,086 messages from 183,303 hanging in the queue.

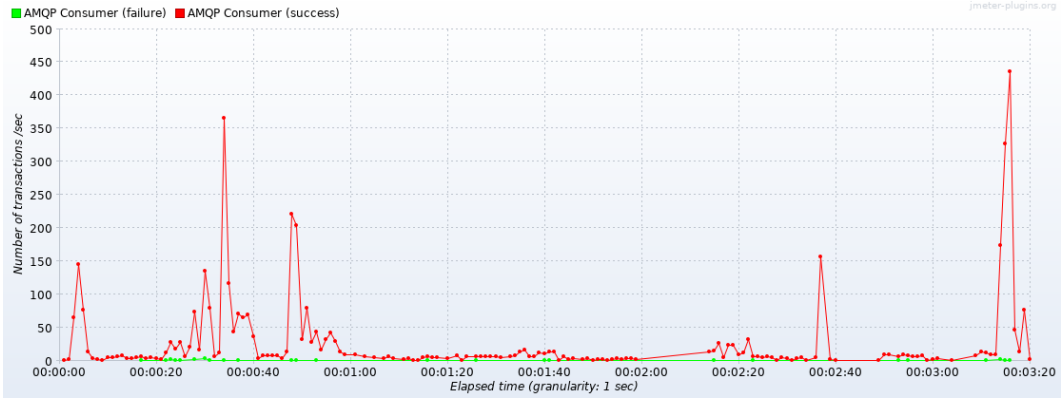

Figure 10 - Subscriber Transactions for Test 1 on the AMQP Scenario

In this trial, we can see that scaling the publishers and subscribers by powers of ten was not ideal. Furthermore, the relationship between the number of publishers and subscribers also needs proper tuning to prevent publishers from clogging the broker, as the 10:1 ratio used in the trials was not ideal for many devices.

3.5. Threats to Validity

Regarding external validity, we acknowledge that our tests are still limited to a small set of devices and technologies, imposing barriers to a broader generalization regarding the applicability of *Kubernetes* on IoT. However, the need to understand the limits of this technology and the particularities of IoT sustains the need for further research. The processor architecture and node resources can be a limiting factor in using Kubernetes in the IoT scenario, as some solutions may not be available for some processor architectures. However, as time goes on, the trend is that embedded devices get more powerful resources.

On internal validity, we consider that all the executed trials were adequately set up to reduce noise and interference from any other factor. The environment where the trials were performed was a clean *Raspbian Linux* operating system with a newly instantiated *Kubernetes* cluster and disabling any network configuration made by *Docker* and using the mainline Docker images for NGINX and RabbitMQ. Furthermore, after each trial, the environment was destroyed and recreated automatically. Therefore, any execution of one trial could not influence another.

To deal with construct validity, we tried to set up the cluster as close to a production environment as possible. All the configurations were set by following the official *Kubernetes* documentation ("Kubernetes," [S.d.]) and each of the respective documentation for the container images. We executed all the trials from an isolated environment and were also parametrized to reflect the usage intensity of a real IoT scenario.

Regarding the evaluation process, we consider that the automated trials generated alongside the configuration files necessary to set up the cluster should enable any person to execute the trials at any time. Furthermore, our implementation source code (including cluster setup configurations, compiled binaries for *Kubernetes* on the *armv6l* platform, trial scripts, and tutorials) are available on an instance of Gitlab[1] and is public. However, as we did not conduct any statistical trials of the results, this represents a conclusion validity threat to our study.

[1] https://www.mrdevops-gitlab.com/plataforma-de-desenvolvimento-continuo-para-iot

3.6. Conclusions

The results present the applicability of *Kubernetes* for at least two scenarios, one using synchronous (*HTTP*) and one using asynchronous (*AMQP*) protocols. During the trials, the cluster kept the applications alive and handled the high load in the *Raspberry Pi*. Furthermore, we believe container-based technologies may enable deployments in highly distributed environments as envisioned by the IoT paradigm.

The adoption of the *Kubernetes* Engine as a platform for IoT still needs more research, as it still imposes barriers regarding the requirements of the cluster worker and primary nodes. Moreover, the cluster implementation may not be suitable for all constrained devices built for IoT purposes in the market. Nevertheless, on platforms that can handle a *Kubernetes* cluster, the gain from deploying and maintaining those applications can outweigh the complexity and overhead of bootstrapping the cluster in those environments.

In future works, we envision improving our trials by using simulations to evaluate how *Kubernetes* would perform in a large-scale scenario. We aim to use Smart City simulation tools and specify a configuration that should enable *Kubernetes* to provision IoT Gateways on-demand.

3.7. Chapter Considerations

In this Chapter, we presented a performance evaluation of Kubernetes as a platform for IoT. To execute the trials, we set up a cluster using three *Raspberry PIs* and a master node on a conventional computer. The results show that while we can use IoT devices as gateways in some scenarios, in a demanding scenario, the lack of hardware capacity may prove to be an impeding factor. Furthermore, there are also limitations regarding the number of available Docker images for the ARM architecture, compatibility with some container network interfaces that do not support ARM, and Kubernetes being a demanding process running on the worker nodes, limiting even further the number of applications that may run in an IoT device.

4 Performance Comparison of Container Orchestration Tools on ARM Platforms

This Chapter presents a performance comparison of container orchestration tools, Kubernetes, K3S, and KubeEdge. For the comparison, we used the same experiment setup presented in Chapter 4.

4.1. Introduction

As discussed in Chapter 2 and Chapter 3, industry practitioners and researchers are trying to adapt already existing container orchestration tools to be used on IoT applications, given that these tools are already proven to work in a conventional software development context.

During this research and the *ad-hoc* literature review, two solutions, besides Kubernetes, caught our attention, K3S and KubeEdge. K3S is a complete replacement for Kubernetes, allowing it to act as a Primary Node and Worker Node in any configuration compatible with Kubernetes. It also wholly adapts the Kube API Server, allowing the same Resource Definitions and Custom Resource Definitions to be used the same way they would on conventional clusters. However, unlike Kubernetes, it does not use *etcd* to store the cluster metadata, replacing it with SQLite, a file-based storage solution that uses SQL-like language.

Unlike K3S, which completely substitutes Kubernetes, KubeEdge replaces Worker Nodes, called Edge Nodes, in KubeEdge terms. KubeEdge, by default, provides an MQTT server using Eclipse Mosquito to enable seamless communication and synchronization of IoT devices while keeping the metadata on each node to allow faster node recovery.

To evaluate these technologies, we set up four cluster configurations, one with Kubernetes, one with K3S, one with Kubernetes as a primary node and KubeEdge as a worker node, and one with K3S as a primary node and KubeEdge as a worker node. We analyzed two scenarios in each configuration, one using the traditional HTTP protocol and another using *MQTT*. This experiment aims to identify if the changes implemented on different orchestrators could impact performance in IoT scenarios.

4.2. Experimental Setup and Procedures

To allow the easy configuration of the cluster and replication of this study, we migrated our setup scripts to use *Ansible* instead of *Chef* to provide all the tools and services on the target platforms. *Ansible* is a cross-platform tool to automate the cloud provision of servers and workstations ("Ansible Documentation — Ansible Documentation," [S.d.]). Unlike *Chef*, which needs a specific executor in each target device, *Ansible* only depends on an existing Python installation to execute and only if particular tasks are being used. To test the scripts and provide an easy way to simulate the deployment environment locally, we also created a *Vagrant* file, a script to provision a set of VMs to allow local deployments of the experiment environment. All the scripts used in this Chapter are publicly available on Github[2].

Regarding the cluster configuration, we used a similar setup to the one discussed in Chapter 3. One virtual machine serves as a primary node, and four *Raspberry PIs, Model 3 B+,* as worker nodes. The primary node is two vCPUs and a 4Gb DDR4 RAM VM running Ubuntu 18.04 LTS. We implemented all the test cases using Apache JMeter version 5.2, and they were executed in a workstation with 24 vCPUs and 64 Gb DDR4 RAM, also running Ubuntu 18.04 LTS. All JMeter scripts are also hosted on GitHub[3] to ensure reproducibility. To ensure the internal validity of the tests, we also made a teardown and a new setup of the cluster after each execution with the help of the installation scripts.

Unlike the previous tests done in this research, the cluster was not physically in the exact location of the activation environment in this experiment, so all the tests were executed from a remote network. Because of that, we could no longer use the same time thresholds of *100ms, 1000ms,* and *10000ms* as used before since all tests would instantly fail because of the inherent latency generated by the distance between the two environments. To mitigate this, we added *200ms*, the value that was the current response time of a simple ping from the activation to the test environment, to all max response time thresholds.

The first test scenario was synchronous for all the cluster configurations. In this scenario, we set up an *NGINX Docker* image version 1.17 from *Dockerhub*, this image

[2] https://github.com/msc-alessandro/ansible

[3] https://github.com/msc-alessandro/JMeter

has a default web page that is 850 bytes in size, and for each request, the cluster should return this page with a 200 HTTP status code and within the time frame for the test to be considered successful. Any other status code, no response from the cluster, or the correct response outside the time frame, would be considered a failure. For this test scenario, the cluster configuration was the same, as shown in Figure 6. However, since we had changed the max response time, Table 4 presents the new values used for the synchronous scenario.

Table 4 - Test Execution Parameters for HTTP Protocol Scenario on Remote Networks

Test ID	RPS	Max Response Time (*ms*)	Thread Number
0	10	300	1
1	10	1200	10
2	10	10200	100
3	100	300	10
4	100	1200	100
5	100	10200	1000
6	1000	300	100
7	1000	1200	1000
8	1000	10200	10000

For the asynchronous tests, we could no longer use *AMQP* since KubeEdge would not support headless services, an essential feature for the *RabbitMQ* cluster deployment in a container orchestrator cluster. To execute the asynchronous test, we decided to keep *RabbitMQ* on the Kubernetes and K3S cluster but use the Eclipse Mosquitto already shipped in KubeEdge for the scenarios that used KubeEdge as worker nodes. Since Mosquitto cannot use the AMQP, we switched to using MQTT on all asynchronous configurations. The test parameters for the asynchronous scenario are not different from those executed in Chapter 3 and are described in Table 2. Figure 13 presents the cluster topology for the Kubernetes and KubeEdge scenarios, close to the K3S and KubeEdge scenarios. Figure 12 shows how the devices outside the network can communicate with KubeEdge nodes using the device mappers using the MQTT

protocol provided by Mosquitto. In the presented situation, these device mappers interface applications and KubeEdge devices, providing an easy way to support multiple devices of different protocols.

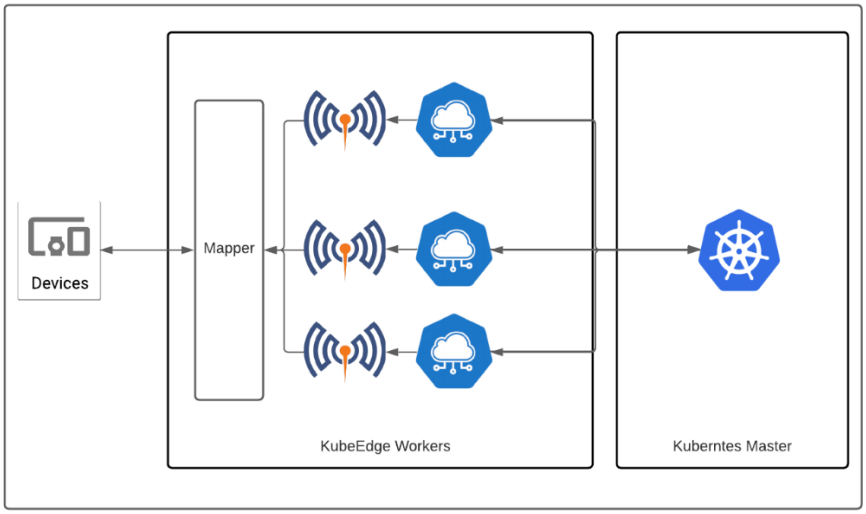

Figure 11 - Cluster Topology for KubeEdge Scenarios

The specific versions of each of the components used in the experiment are described in Table 5.

Table 5 - Software Versions for Each of the Components used in the Experiment

Component	Version
Kubernetes	1.18.12
K3S	1.18.12+k3s1
KubeEdge	1.5.0
SQLite	3.33.0
Containerd	v1.3.3-k3s2
Flannel	v0.11.0+k3s.2
Metrics Server	v0.3.6
Traefik	1.7.19
CoreDNS	v1.6.9

Local Path Provisioner	v0.0.1.1

4.3. Results

The first of the analyzed technologies was *Kubernetes*, as its' performance can be considered the baseline for the experiment. After the cluster creation and the improvements on the *JMETER* execution scripts, we executed the tests on the cluster that was now on the Federal University of Rio de Janeiro (UFRJ) infrastructure. The first tests performed were the synchronous scenario with the *HTTP* protocol. As demonstrated by Table 5, the cluster could handle requests from one hundred threads, achieving a throughput close to a thousand requests per second.

Table 6 - Test Execution Results from the HTTP Scenario using Kubernetes

Test ID	Samples	Mean (*ms*)	Min (*ms*)	Max (*ms*)	Median (*ms*)	Error %	Throughput (transactions/sec)
0	1286	139,63	135	297	137,00	0,00%	7,15
1	1802	140,45	135	299	138,00	0,00%	10,01
2	1889	161,96	135	1277	140,00	0,00%	10,49
3	12632	138,88	134	297	137,00	0,00%	70,15
4	17990	139,49	134	1296	137,00	0,00%	99,88
5	18895	166,31	134	3317	139,00	0,00%	104,78
6	127032	137,84	134	318	136,00	0,00%	705,35
7	180019	138,88	134	3321	136,00	0,00%	998,00
8	81643	19853,84	0	131720	281,00	24,17%	268,35

However, in test number 8, the high number of requests clogged the clusters, reducing its' throughput to 268 requests per second. In this last test case, 24% of all samples failed, mostly with connection timeout received from the cluster, as presented in Figure 13. During the same test, the cluster mainly became non-operation for user interaction and commands because of the high memory usage of *NGINX pods* and the

inability of Kubelet to run correctly on the worker nodes. Nonetheless, after giving the cluster some time for self-healing, all the nodes became operational again.

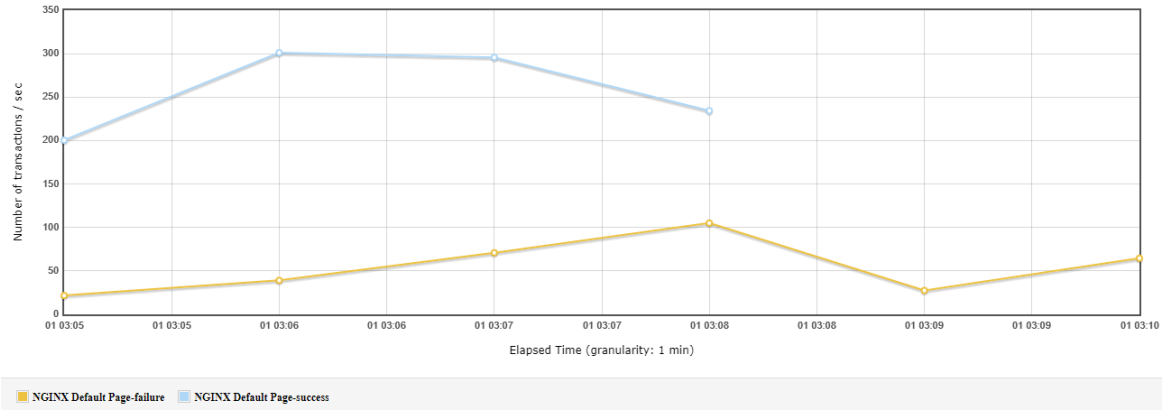

Figure 12 - Kubernetes HTTP Test 8 Transactions

Using the dataset generated by JMeter, we also created a boxplot comparing all nine test configurations. This allowed us to analyze the outliers on tests with IDs 7 and 8. The boxplot allowed us to visualize that the results were close for most tests, with low variability. However, during the tests with the high number of threads, there were a lot of outliers, ranging from 10000ms to 100000ms. It shows the moments when the cluster was stuck, especially because of the large number of requests.

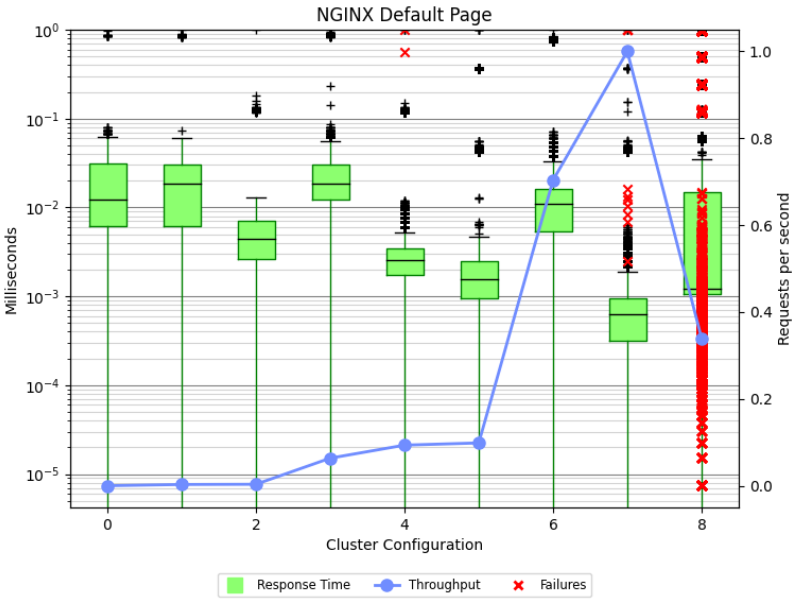

Figure 13 - Boxplot for the K8S HTTP Experiment

Following the round of tests with Kubernetes, we proceeded to test K3S. As said before, K3S is simpler than Kubernetes, having an easy installation process with only one binary file, and being based on a file-based database, instead of an in-memory database. For these tests and to provide a meaningful comparison between technologies, all the technologies, and parameters used in Kubernetes were also used here, including container runtime, as Docker was used in both tests.

For K3S, the results did not differ significantly from Kubernetes in any scenario, even with the system's differences in storage and services. Compared from a performance standpoint, K3S could substitute Kubernetes in a cluster without disruption or loss. According to K3S documentation, the drawback from Kubernetes shows up when considering a cluster with a high number of nodes. Specifically, K3S could suffer from IO limitations when using the disk to store cluster metadata. However, the performance was not impacted in a small cluster with Raspberry PIs and fast enough SD Cards. Table 6 below summarizes the results of the HTTP tests using K3S.

Table 7 - Test Execution Results from the HTTP Scenario using K3S

Test ID	Samples	Mean (*ms*)	Min (*ms*)	Max (*ms*)	Median (*ms*)	Error %	Throughput (transactions/sec)
0	1294	138,76	134	296	137,00	0,00%	7,19
1	1803	141,00	135	296	138,00	0,00%	10,01
2	1889	160,96	135	340	139,00	0,00%	10,49
3	12644	138,72	134	291	137,00	0,00%	70,21
4	17990	138,95	134	293	137,00	0,00%	99,89
5	18908	163,45	134	3291	138,00	0,00%	104,84
6	126582	138,33	134	299	137,00	0,00%	702,86
7	180034	138,69	134	1330	136,00	0,00%	998,20
8	82474	19505,94	0	131537	291,00	24,06%	267,50

Comparing the results of both technologies, in all scenarios, the differences were at most 1%, with K3S being better in the max response times for tests 2, 4, 6, and 7. However, on many requests, the cluster behaved like Kubernetes, rejecting connections

and putting the cluster in a frozen state. By analyzing the transactions per second of both clusters in Figure 14, we can see that K3S started failing and has a faster degradation in terms of the number of failures per second and a sharper decline in the number of successful transactions.

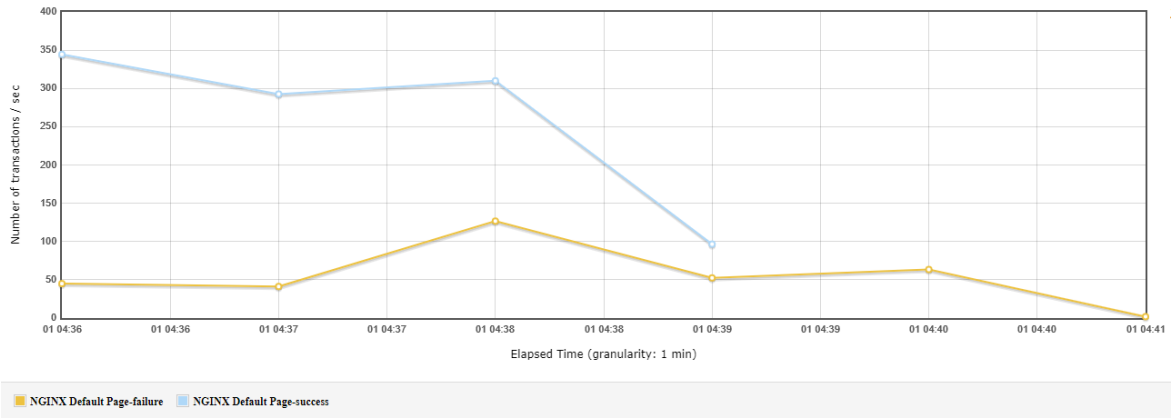

Figure 14 - K3S HTTP Test 8 Transactions

Analyzing the boxplot generated for the K3S HTTP tests and comparing it with the one in Figure 13, we can see that both cluster configurations did not change regarding the spread of outliers since they were mostly the same. However, K8S had a higher average throughput during Test ID 6, almost four times the throughput of K3S. Figure 15 presents the boxplot for K3S.

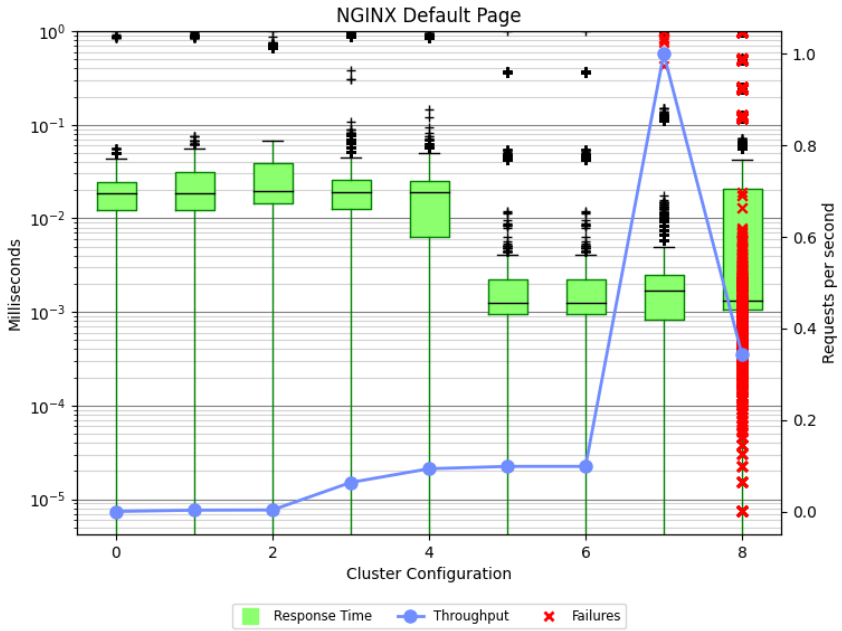

Figure 15 - Boxplot for the K3S HTTP Experiment

After finishing the tests with Kubernetes and K3S, we proceeded to do the same tests, now with KubeEdge as worker nodes in both configurations. Initially, we prepared to set up KubeEdge in the same fashion as the other experiments. However, the KubeEdge proposal differs from the different technologies as it focuses on the worker nodes and allows the usage of protocols more suited to IoT use cases, such as MQTT and, in the future, LoRa. However, at present, KubeEdge is still lacking many features that the other two technologies have, such as the presence of a *Container Network Interface (CNI).* Without said component, the intra-cluster communication between services is impaired, not allowing, for example, the usage of service discovery endpoints necessary for many applications. The development of Edge Mesh, a service mesh for Edge Nodes, is in the current roadmap for KubeEdge, but at the time of this study, it is not released to the public.

Since the first experiments, the use of NGINX did not depend on service discovery technologies. Therefore, the test followed the same principles as Chapter 3 and the other experiments regarding Kubernetes and K3S. However, during the setup of the asynchronous test, we could not replicate the same *RabbitMQ* cluster on KubeEdge because of the networking limitation. Therefore, we decided to bypass that limitation using the *Mosquitto Server* already present on the KubeEdge implementation.

As for the results, KubeEdge showed similar performance using Kubernetes and K3S as masters and even using Kubernetes and K3S as standalone. In the first test, with Kubernetes working with KubeEdge, the tests show that the max execution time was higher when using KubeEdge than other cluster configurations. Furthermore, during tests 0 and 1, KubeEdge also presented timeout errors, while previous tests showed none. However, even with the given errors, in the long run, KubeEdge showed the capacity to run the same environment as Kubernetes with differences regarding throughput and errors below 1%. Table 7 presents the results of the tests run with Kubernetes and KubeEdge.

Table 8 - Test Execution Results from the HTTP Scenario using Kubernetes and KubeEdge

Test ID	Samples	Mean (*ms*)	Min (*ms*)	Max (*ms*)	Median (*ms*)	Error %	Throughput (transactions/sec)
0	1295	138,71	135	301	137,00	0,08%	7,19

1	1802	140,66	135	1284	138,00	0,06%	10,01
2	1889	164,11	135	3299	141,00	0,00%	10,49
3	12621	139,01	134	299	137,00	0,00%	70,10
4	17988	139,63	134	315	137,00	0,00%	99,88
5	18906	161,40	134	1300	138,00	0,00%	104,82
6	126614	138,29	134	1298	137,00	0,00%	702,94
7	180040	139,05	134	3321	139,00	0,00%	998,18
8	82526	19951,92	0	141838	302,00	24,53%	273,02

KubeEdge presented a similar behavior concerning throughput curves with K3S, with steeper performance losses when the number of threads and requests is high. Figure 15 illustrates the transactions per second achieved by Kubernetes and KubeEdge in test 8.

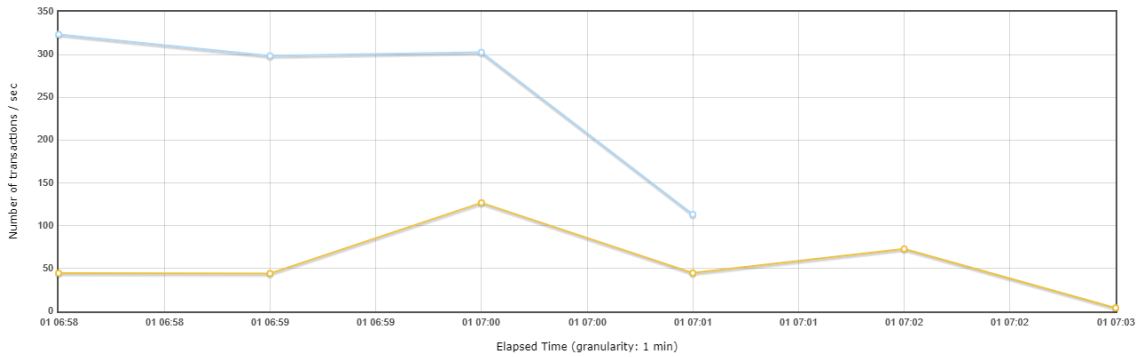

Figure 16 - Kubernetes and KubeEdge HTTP Test 8 Transactions

As with Kubernetes, the tests result for the configuration K3S, and KubeEdge did not show differences more significant than 1% in throughput and failures across all test cases. This scenario also has the same pattern of worsening the max response times. However, the median is roughly the same for all test cases across any configuration. Table 8 presents the data for the tests in the K3S and KubeEdge configurations.

Table 9 - Test Execution Results from the HTTP Scenario using K3S and KubeEdge

Test ID	Samples	Mean (*ms*)	Min (*ms*)	Max (*ms*)	Median (*ms*)	Error %	Throughput (transactions/sec)
0	1271	141,37	135	291	140,00	0,00%	7,06
1	1803	142,47	135	292	140,00	0,00%	10,01
2	1889	164,53	134	1297	142,00	0,00%	10,49
3	12527	140,08	134	293	139,00	0,00%	69,57
4	17988	139,98	134	1303	137,00	0,00%	99,88
5	18892	167,31	134	7521	139,00	0,00%	104,74
6	127011	137,86	133	1304	136,00	0,00%	705,21
7	179986	138,80	133	7538	136,00	0,01%	997,89
8	80838	20962,52	0	131578	296,00	24,08%	268,39

Analyzing Figure 16, we can see that the only scenario where the cluster degradation happens systematically is when only Kubernetes is running. It can be explained that Kubernetes is more robust and mature than other technologies and can store data in memory, contrary to the different technologies, where accessing the disk during demanding scenarios can cause overhead on the cluster operation.

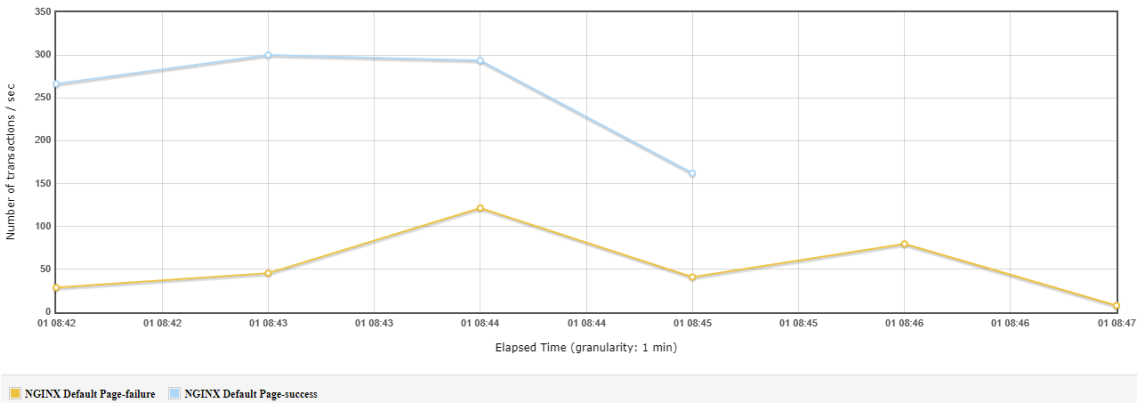

Figure 17 - K3S and KubeEdge HTTP Test 8 Transactions

We also compared all four clusters in the experiment with ID 8 since it had the highest number of outliers and errors. The boxplot presented in Figure 18 shows K8S, K3S, and K8S with KubeEdge and K3S with KubeEdge, respectively. Based on this graph, we can see that K3S with KubeEdge had the lowest average throughput under heavy load than any other cluster configuration. However, the outliers were consistent between the tests, even when using completely different technologies.

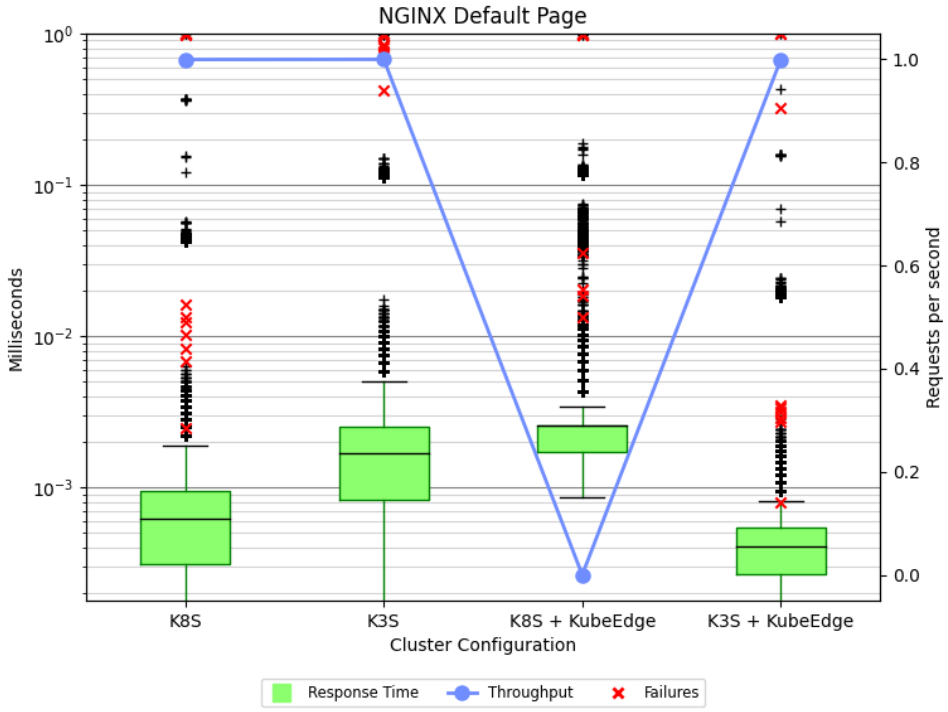

Figure 18 - Comparison of Test 7 between K8S, K3S, K8S with KubeEdge and K3S with KubeEdge

Following the tests with the synchronous scenarios, we proceeded to the asynchronous protocol, using MQTT instead of AMQP. The first cluster configuration was our baseline, Kubernetes. In this test scenario, we did not specify a max response time for failures; however, we did specify that the number of subscribers should always be ten times the number of publishers to mimic an IoT scenario. The values for each test case are like in Table 2 in Chapter 3. Finally, in Table 9, we present the results regarding the test scenarios on Kubernetes.

Table 10 - Test Execution Results from the MQTT Scenario using Kubernetes

Test ID	Description	Samples	Mean (*ms*)	Min (*ms*)	Max (*ms*)	Median (*ms*)	Error %	Throughput (transactions/sec)

0	MQTT Pub Sampler	12727	137,64	134	190	137,00	0,00%	70,88
0	MQTT Sub Sampler	12718	0	0	0	0	0,00%	70,98
0	MQTT Pub Connections	10	312,20	289	484	293,00	0,00%	1,08
0	MQTT Sub Connections	1	484,00	484	484	484,00	0,00%	2,07
1	MQTT Pub Sampler	126617	138,06	134	542	138,00	0,00%	704,98
1	MQTT Sub Sampler	832032	0,00	0,00	0,00	0,00	0,00%	4659,96
1	MQTT Pub Connections	100	307,24	281	1362	288,00	0,00%	9,88
1	MQTT Sub Connections	10	410,30	22	1498	289,50	0,00%	1,08

It is possible to see that using a lightweight protocol such as MQTT, the throughput of the cluster is greatly improved, as, in the test with 100 publishers and ten consumers, the aggregated throughput is over five thousand transactions per second. Nevertheless, even with the high throughput, the cluster did not present any failure and was fully responsible during the testing. However, in the same way that happened for the synchronous scenarios, there were no significant differences in results when comparing Kubernetes with K3S, as in all test cases, the difference between the two technologies was less than 1% in throughput and response times. Ultimately, we acknowledge that the choice between these two clustering technologies can be based on cluster size, as K3S depends on using an external database to achieve high availability and complexity of the desired cluster. However, performance-wise, the two technologies presented the same results regarding throughput and response times. Table 10 below shows the results for the K3S MQTT test scenarios.

Table 11 - Test Execution Results from the MQTT Scenario using K3S

Test ID	Description	Samples	Mean (*ms*)	Min (*ms*)	Max (*ms*)	Median (*ms*)	Error %	Throughput (transactions/sec)
0	MQTT Pub Sampler	12727	137,69	135	241	137,00	0,00%	71,10
0	MQTT Sub Sampler	12706	0	0	0	0	0,00%	71,14
0	MQTT Pub Connections	10	412,70	290	1491	292,50	0,00%	1,08
0	MQTT Sub Connections	1	473,00	473	473	473,00	0,00%	2,11
1	MQTT Pub Sampler	126414	138,27	135	203	138,00	0,00%	703,86
1	MQTT Sub Sampler	676789	0,00	0,00	0,00	0,00	0,00%	3774,55
1	MQTT Pub Connections	100	318,08	282	1455	289,50	0,00%	9,88
1	MQTT Sub Connections	10	310,50	283	497	288,50	0,00%	1,08

The next group of tests used the KubeEdge configuration for worker nodes, as described in the synchronous scenarios. Using this configuration, *Mosquitto* achieved results close to the Kubernetes and K3S standalone clusters, with variations in the range of 1%. In this test, the same considerations made by KubeEdge apply, as it does not implement the same features of Kubernetes and K3S. Table 11 below presents the test results in Kubernetes and KubeEdge configuration. The presented results show that KubeEdge may be used as an MQTT cluster, as it has features that allow the usage of such protocol and sync the information between all nodes without the need for more complex services such as *RabbitMQ,* however in scenarios where the complexity of service discovery is needed, or technologies are vendor locked, KubeEdge can be disadvantageous.

Table 12 - Test Execution Results from the MQTT Scenario using Kubernetes and KubeEdge

Test ID	Description	Samples	Mean (*ms*)	Min (*ms*)	Max (*ms*)	Median (*ms*)	Error %	Throughput (transactions/sec)
0	MQTT Pub Sampler	12775	137,16	134	201	137,00	0,00%	71,15
0	MQTT Sub Sampler	12764	0	0	0	0	0,00%	71,23
0	MQTT Pub Connections	10	301,90	278	486	281,50	0,00%	1,08
0	MQTT Sub Connections	1	486,00	486	486	486,00	0,00%	2,06
1	MQTT Pub Sampler	127479	137,11	134	531	136,00	0,00%	709,86
1	MQTT Sub Sampler	826535	0,00	0,00	0,00	0,00	0,00%	4629,26
1	MQTT Pub Connections	100	315,62	270	3523	277,00	0,00%	9,89
1	MQTT Sub Connections	10	402,20	270	1509	279,50	0,00%	1,08

Following the tests with KubeEdge, the fourth test setup used K3S as the master node. It is because the KubeEdge documentation ("What is KubeEdge — KubeEdge Documentation 0.1 documentation", [S.d.]) recommends using K3S as the master for the cluster. However, we could not notice any performance changes when using Kubernetes or K3S. As in the other tests, the variation across tests in configurations using KubeEdge was within 1% of performance, as shown in Table 12 below.

Table 13 - Test Execution Results from the MQTT Scenario using K3S and KubeEdge

Test ID	Description	Samples	Mean (*ms*)	Min (*ms*)	Max (*ms*)	Median (*ms*)	Error %	Throughput (transactions/sec)

0	MQTT Pub Sampler	12593	139,04	134	178	139,00	0,00%	70,41
0	MQTT Sub Sampler	12584	0	0	0	0	0,00%	70,47
0	MQTT Pub Connections	10	405,80	276	1494	284,50	0,00%	1,08
0	MQTT Sub Connections	1	480,00	480	480	480,00	0,00%	2,08
1	MQTT Pub Sampler	127850	136,72	133	500	136,00	0,00%	711,85
1	MQTT Sub Sampler	831698	0,00	0,00	0,00	0,00	0,00%	4640,71
1	MQTT Pub Connections	100	309,45	269	1504	280,00	0,00%	9,88
1	MQTT Sub Connections	10	298,90	271	505	276,50	0,00%	1,08

4.4. Threats to Validity

Regarding external validity, we acknowledge that the small subset of scenarios executed in the tests presented in this Chapter do not represent the totality of scenarios where Kubernetes, K3S, and KubeEdge can be used and are comparable. Therefore, our results only make assertions on the performance of such technologies in the presented scenarios based on that assumption. Further tests are needed to evaluate other scenarios regarding other aspects and possible configurations of these technologies.

On internal validity, we consider that all the executed tests were adequately set up to reduce the influence of any confounding factors. All tests were performed from 1:00 AM to 5:00 AM (GMT-3) to minimize network overhead since the cluster was physically distant from where the tests were running. In addition, after each test case, the cluster was recreated not to allow cross-influence between the test cases.

To deal with construct validity, we set up the cluster configurations by following the official Kubernetes, K3S, and KubeEdge documentation and each respective documentation for the container images used in the experiments. We executed all the tests from an isolated environment and were also parametrized to reflect the usage intensity of a real IoT scenario.

Regarding the evaluation process, we improved the setup scripts for all cluster configurations compared to the experiments in Chapter 3. All scripts are now written using *Ansible* and *Vagrant* for a more straightforward configuration of a local or cloud cluster of any technologies presented in this Chapter. The scripts with all the files needed to execute the same experiment on Windows Hyper-V and any Linux Distribution are hosted on Github.

4.5. Conclusions

The presented results show that all container orchestrators tested can be used to set up a cluster on Raspberry PI model 3 B+. However, more mature solutions, like K3S and Kubernetes, are easier to set up and configure in these situations. At the same time, KubeEdge brings more complexity to the worker nodes, and its benefits are still unclear.

Regarding performance, all the presented solutions had similar results, with only KubeEdge being slower to respond in some HTTP cases caused by the overhead that its master components bring to the primary node. Furthermore, KubeEdge can also not run all Kubernetes components, Container Network Interfaces (CNI), and KubeProxy, making it impossible for applications that depend on service discovery endpoints such as RabbitMQ on cluster mode. However, when considering all the processes of setting up the cluster and realizing the tests, K3S is the simplest, as it is distributed as a single binary file executed on the master and worker nodes.

As the results of the direct comparison of all technologies in this study, we firmly believe that KubeEdge is not yet fit for production environments, as it is still in early development with many features missing from its competitors. However, when comparing K3S and Kubernetes, both technologies may be used in production environments, as applications developed for one is automatically compatible with the other as K3S implements all features from Kubernetes. The one drawback of K3S is currently scalability, as it needs a relational database combined with a fixed registration

address for the worked nodes to register against to be configured to High Availability (HA).

4.6. Chapter Considerations

In this Chapter, we presented a performance comparison of three container orchestrator platforms, one designed for the cloud and the other for IoT scenarios. To execute the experiment, we set up two scenarios, totaling ten test cases for each cluster configuration and forty tests. While K3S presented some advantages over Kubernetes, such as easy installation and configuration and low memory footprint, the performance of the two was not different. However, in the scenarios using KubeEdge, the performance was impacted, as the tests have shown a lower performance for HTTP scenarios. KubeEdge is also much harder to set up and configure than all the other solutions and, in the actual development state, is incompatible with the same resource definitions as Kubernetes and K3S.

Based on the presented tests in Chapters 3 and 4, we could conclude that it is possible to use container orchestration tools as a deployment platform for IoT applications. Of course, we know that each container orchestration tool has pros and cons; however, in the case of K3S, the ease of setup and the native compatibility with the ARM64 devices make it a good choice for cluster creation proposed architecture.

Using K3S, we constructed a cluster of Raspberry PIs 3 in the LENS/ESE laboratory. With the cluster, we set up our proposed architecture in Chapter 5 to deploy applications on the Raspberry PI 3. During two years of development and deployment, we could understand the weaknesses and strengths of using such a platform.

5 Architectural Proposal for Continuous Development on IoT Devices

Based on all the tests on ARM devices' performance and container orchestration technologies, we propose a possible architecture for the continuous development of IoT devices in this chapter. The architecture uses the same principles discussed in the previous chapters and integrates them with Gitlab, an open-source code management platform. Then, we show results on how we used this architecture to build and deploy applications in the context of the Experimental Software Engineering Laboratory in COPPE/UFRJ.

5.1. Introduction

Based on the results of the three performance analysis executed in Chapters 3, 4, and 5, we indicated that Kubernetes and its resources could be used in the context of IoT applications. These findings corroborate some articles already found in the literature, such as the one by (HUANG et al., 2019), and allowed us to propose an architecture for the development of IoT applications using Kubernetes as a means to deploy an application on Raspberry PIs, specifically using Raspberry PI to run worker nodes for Kubernetes.

The architecture presented in this Chapter was then built in the Software Engineering Laboratory (LENS) at the University of Rio de Janeiro and used by students in other groups who wanted an environment to test and deploy their IoT applications. Currently, the implementation of such architecture is still in use by some of the IoT software systems developed at the university.

5.2. Proposed Architecture

The proposed architecture considers an automation tooling that allows developers to contribute, build, test, and deploy their code in a cluster. The platform used to contribute and manage the generated code on the projects, we decided to use Gitlab, an open-source code management tool that identifies itself as a DevOps platform. Gitlab uses Git in its backend, allowing users to control all alterations done to the code. It also has features regarding activity management, service desk, team planning, and requirements management (“GitLab Documentation,” [S.d.]). However, in the context of

this proposed architecture, the relevant feature is the capacity to create automated pipelines to allow code pushed to the platform to be built, tested, and deployed in some environment.

The platform achieves this capability by using components called Gitlab Runners. Practitioners can deploy these runners in the environment where they should interact and receive calls from the Gitlab Server to execute the commands to build the application, test it, and any other tools that may be necessary to produce the final artifact. Kubernetes can run the Gitlab Runners or any compatible tool, in the cluster, by deploying a group of software containers responsible for taking the pipelines and executing them.

Figure 17 shows that the proposed architecture presents all the tools necessary to provide features like building, testing, and continuous deployment. It also offers the interaction between the Gitlab Runners and Kube API Server, sending commands through the command line to allow the deployment of all the services that will run on the cluster.

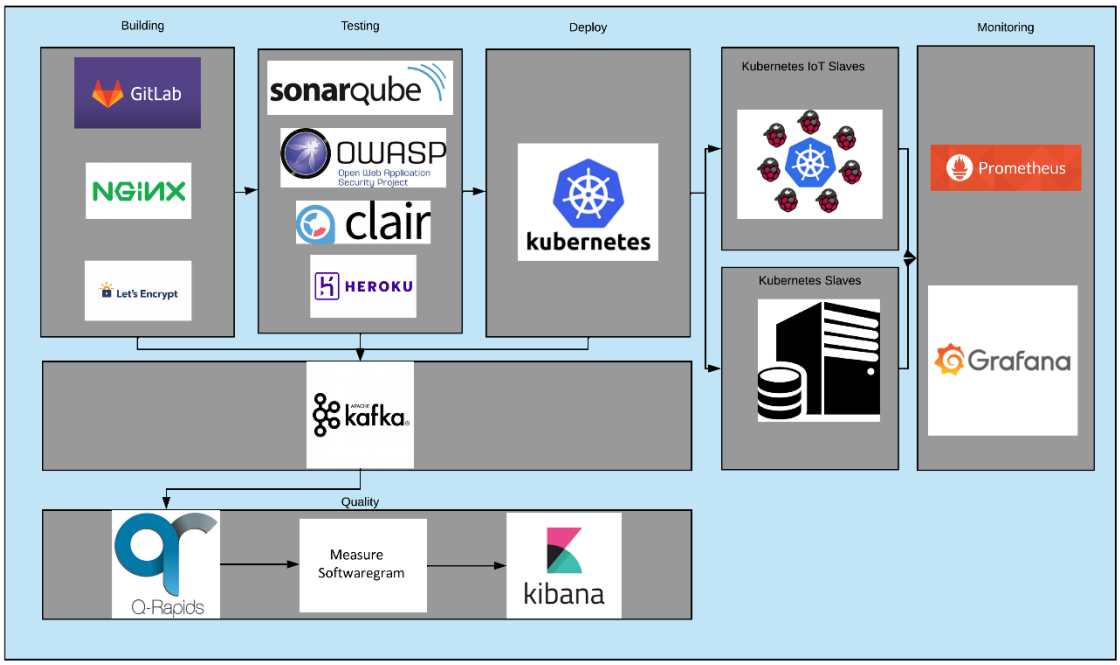

Figure 19 - Proposed Platform for IoT Continuous Development

Based on Figure 19, the architecture components are:

- **Gitlab**: Code management platform has been used chiefly for developers to share and control changes in their code; it is also responsible for triggering jobs that will run on the runners.

- **Gitlab Runners**: The runners responsible for executing the jobs in the target machines are defined based on a configuration file in each code repository.
- **Kubernetes Master**: The node executing the Kubernetes master, the Gitlab Runners, will interact with it using the Kube API Server to schedule deployments in the worker nodes.
- **Kubernetes Worker Nodes**: The nodes executing the Kubernetes and acting as worker nodes. These nodes do not need to run all the services for Kubernetes, but they are the destination of containers that will run the applications at the end.

5.3. Environment Building

Based on the proposed architecture described in Chapter 5.2, we installed the toolset of Gitlab, Gitlab Runners, and Kubernetes on LENS. However, because of technical and hardware limitations, we needed to adapt how the runners interacted with Kubernetes.

The first technological decision regarding the cluster is to decide which container orchestration tool will be used on the master and worker nodes. Based on the experiments in Chapters 3 and 4, it is possible to see that there is no noticeable difference in performance in synchronous or asynchronous scenarios in all analyzed container orchestration tools; however, regarding configuration and maintainability, there are differences between Kubernetes, K3S, and KubeEdge. We did an overview of these three technologies in Chapter 2, but there are more characteristics that we can list here as deciding factors when deciding the technologies we will use. Some of these characteristics are:

- **Kubernetes**
    - It is the most mature solution when compared to the other ones.
    - Regarding the setup of the solution, Kubernetes is the most complex of the three since all services must be installed by the practitioner when using bare-metal clusters.
    - It is compatible with most market solutions, including container network interfaces, ingress controllers, and service discovery tools.
    - It is, by default, the most resilient of the three services since Google designed Kubernetes for large-scale deployments.

- **K3S**
    - It is based on Kubernetes and can completely substitute the Kubernetes components.
    - By default, SQLite, a file-based database, is used instead of etcd, an in-memory database, to store cluster configuration.
    - Rancher Labs distribute K3S as a single binary instead of multiple services to facilitate the deployment and configuration of services.
    - It Includes an already configured network interface and ingress controller to allow an easy setup of those services.
    - It is compatible with service discovery, allowing services like RabbitMQ to work as a cluster.
- **KubeEdge**
    - KubeEdge is the newest of the three solutions and is more immature.
    - It has an entirely different architecture since it depends on Kubernetes or K3S masters to work since it only substitutes the components of the worker nodes.
    - It is not compatible with container network interfaces already on the market, forcing the user to use the included network interface.
    - Since it uses a container network interface designed by the KubeEdge team, it does not allow the installation of any other interface.
    - It includes a Mosquitto MQTT cluster but does not allow the installation of other MQTT clusters, such as RabbitMQ.

Using all the information gathered regarding these container orchestration tools and based on the performance evaluations, we decided to use K3S as the node orchestration tool. It generates less memory pressure on the nodes since it does not use etcd as data storage and depends on fewer service installations as it is a single binary file on both master and nodes.

After the decision to use K3S as the container orchestrator for the environment, we started configuring other tools. The Coppe/UFRJ Infrastructure Team provided us with two virtual machines for installing the Gitlab Server and K3S master. The virtual machines for these tools are necessary because Gitlab and the K3S master can consume many resources. The hardware requirements for Gitlab Server and K3S are:

- **Gitlab Server**
    - CPU: Recommended is four cores.
    - Memory: 4GB is the required minimum.
    - Disk: The installation package is 2.5 GB, but we must consider all the repositories and compiled solutions for disk space.
- **K3S**
    - CPU: Recommended is one core.
    - RAM: 512MB is the required minimum, and the recommended is 1GB.
    - Disk: The binary is 56.9 MB, but we must consider the size of container images and possible mounted volumes for applications running.

Another adaptation needed for the build and deployment pipelines is a Gitlab Runner running on the master node and another on the worker node. Ideally, we would need a set of build machines capable of communicating with the Kube API Server and another set running ARM64 processor architecture to allow the built containers to be run on the Raspberry PI 4 and 3. Figure 20 describes the current architecture running in the laboratory (LENS/ESE).

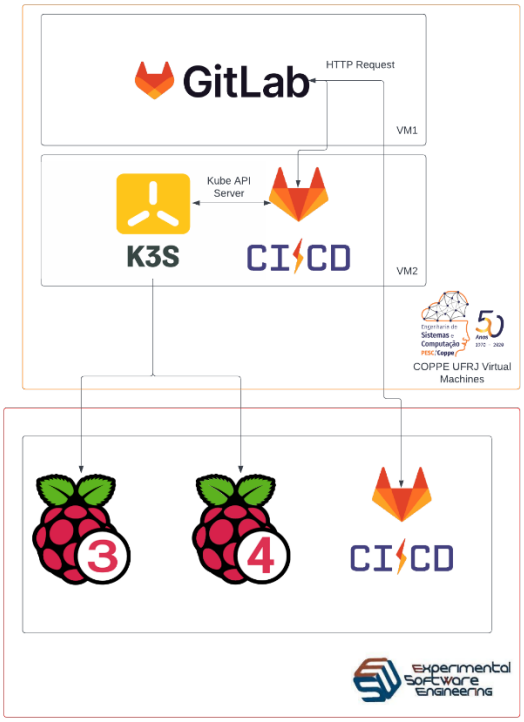

Figure 20 - Architecture Implemented on LENS/ESE Laboratory

The architecture presented in Figure 19 was implemented in the cluster using the layout proposed in Figure 20. Two Raspberry PIs were used as builders since we needed to build the container images in an ARM device. These devices also had several tools to execute automated tests and scans of the container and the code under build. Figure 21 below presents how the services were distributed on the devices.

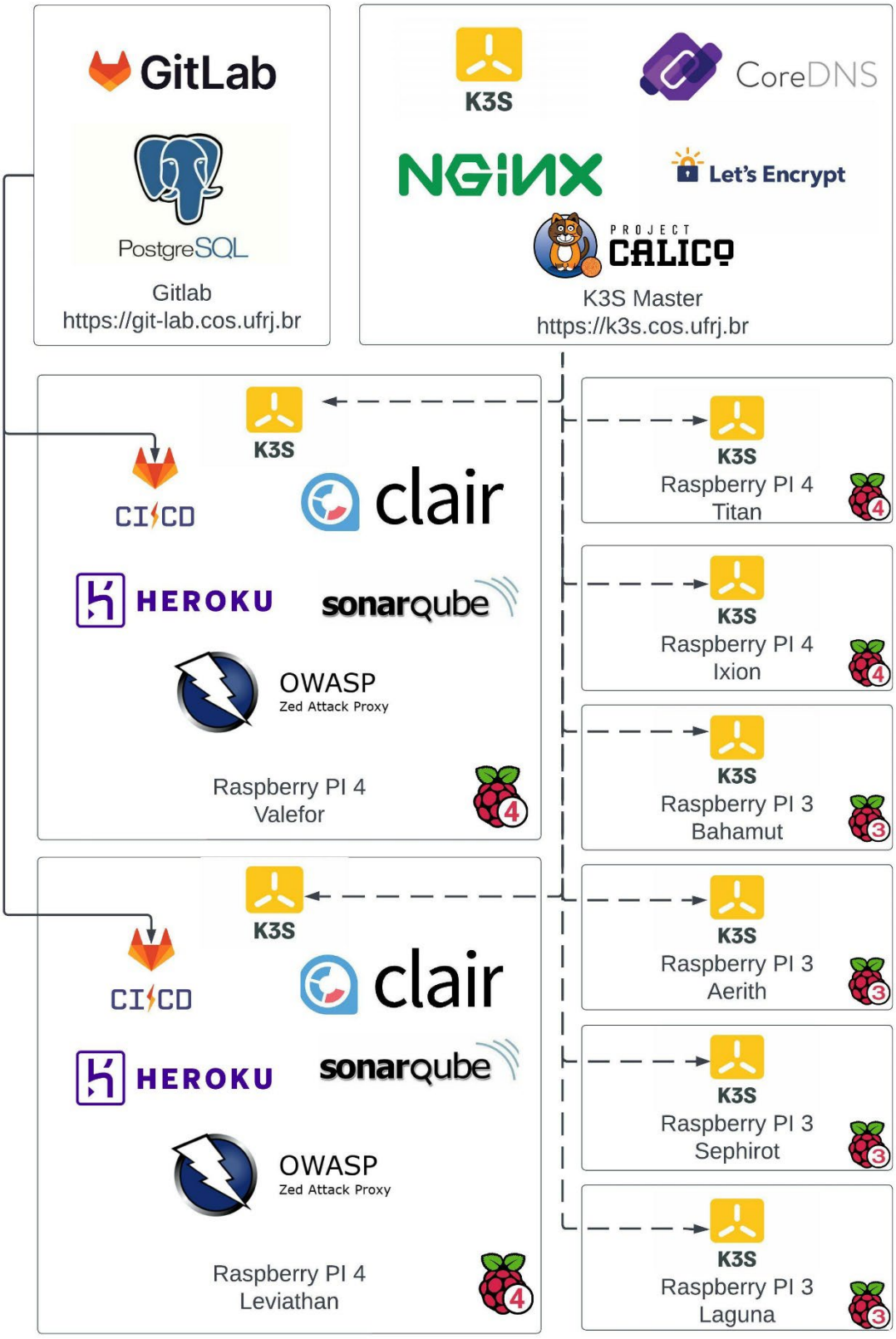

Figure 21 - The Implemented Architecture on The Cluster

In the architecture presented in Figure 18, the dotted lines represent the communication between the master and worker nodes using the overlay network of K3S. The continuous lines represent the communication between the Gitlab Server and the Runners on the Raspberry PI. We describe each of the tools in the builders below:

- **Gitlab Runners**: These are the components responsible for receiving the build definition and triggers from the Gitlab Server and executing them.
- **Clair Container Scanner**: An open-source automated security scanning tool for containers built by Red Hat, allowing anyone to execute and generate reports based on the most recommended actions based on a customizable ruleset.
- **Heroku Buildpacks**: A project initially developed by Heroku and later published as an open-source tool. It allows the creation of Docker container images of applications that initially do not have Dockerfiles or already built images.
- **SonarQube:** A static application scanning tool that detects code smells and refactoring points in the code. It is used as a means to code quality and generates reports based on the code on the pipeline.
- **OWASP Zed Attack Proxy:** A command-line tool to generate and evaluate attacks to APIs. In the context of the proposed architecture, it can be used in a release pipeline to test and detect vulnerabilities on de exposed endpoints of the cluster and its applications.

5.4. Regarding the ARM Architecture Devices as Deployment Targets

While the developers and teams used the cluster to deploy their applications, it became clear that a minimum understanding of the target architecture was necessary to prevent conflicting executables and build artifacts to be scheduled to ARM nodes. At the start of the development of the applications that we will discuss in the next section, most of the developers did not know of the specifics and architecture of the cluster they were developing on, which caused many problems regarding the usage of customized tools and scripts that did not take in consideration the cluster architecture.

To prevent errors from happening because of the different processor architectures between the target devices and the development environment, we decided that all builds would occur in one of the nodes used in the cluster. To do that, a Gitlab

Runner was installed in two of the Raspberries, with all the tools necessary to build the applications under development. Of course, using cluster nodes as build environments may impact the performance of applications already running on the same node. However, since we did not have a dedicated build infrastructure, it was the only way to guarantee that all images produced had the same architecture as the cluster nodes.

Other limitations regarding the usage of ARM processors are the number of images on DockerHub that are compatible with that architecture. Nowadays, most famous tools and services are compatible with ARM. However, some applications still don't have an ARM-compatible image, for example, MySQL.

5.5. Application Characterization

During the implementation of the proposed architecture, groups of developers working under the hood of the LENS/ESE laboratory and at the Hospital Complex of UFRJ began to develop applications during 2020-2022. Using these applications, we could test if the proposed architecture could attend the deployment of these applications in a Kubernetes cluster.

It is important to note that not all applications deployed during this period are IoT. During these years, five applications used the proposed architecture to deploy their apps in the laboratory environment; from the five applications, three composed alternate settings for one specific application. SisCEATE, the word in Portuguese (Sistema Central de Apoio à Saúde dos Trabalhadores e Estudantes da UFRJ), is one of the applications used in this observational study). It is a monolithic application developed in PHP and used to attend to workers and students from UFRJ. SisCEATE has three environments currently running in the cluster; that is why it is considered more than one application.

The other application deployed to the cluster is SAFE, a software developed by students of the Software Engineering graduation course of the UFRJ to monitor the usage of UFRJ installations during the pandemic. SAFE is composed of three services: backend, frontend, and dashboard. It also uses a MySQL database and RabbitMQ as a broker.

5.6. Results

The application deployment pipeline we will address first is SisCEATE. This application is not IoT; however, it was the one that mainly used the cluster to deploy its services. During all the time that SisCEATE was using the platform, developers pushed code 290 times over two years, and during the same period, the release was triggered

151 times. The pipeline for this application consists of four stages: build, rollout, migrate, and seed. Each of the steps does the following:

- **Build**: The software container of the application is built inside the Raspberry Pi to ensure that the architecture used is ARM64, then the container is pushed to the GitLab Registry.
- **Rollout**: The runner on the master node schedules a rollout of the new image to the nodes currently hosting the pods; K3S then pulls the new image and updates the pods with the latest version of the application.
- **Migrate**: In this step, the runner schedules a Kubernetes Job to update the database schema.
- **Seed**: In this step, the runner updates the initial database data; this step only happens when the initial data is not in the database.

Each successful pipeline run creates a new software version, and Gitlab triggers a pipeline each time a developer pushes code to the repository. During the two years of development, there were 151 pipeline executions; however, only 106 were successful, a success ratio of 72.60%. Figure 22 presents all pipelines running from 08/08/2021 to 08/08/2022.

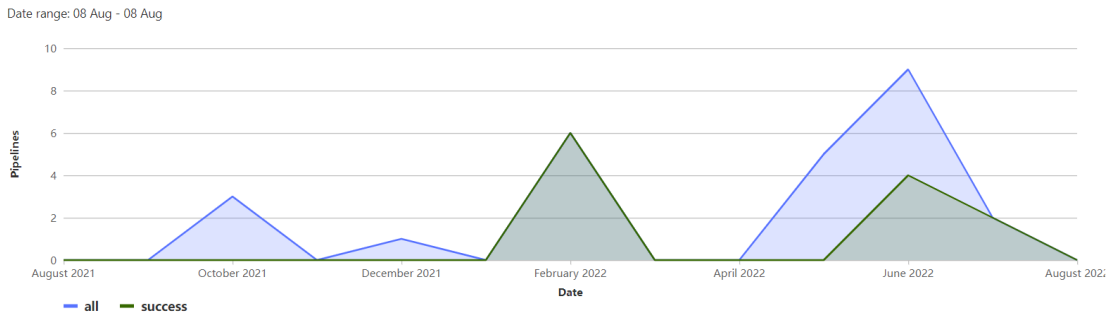

Figure 22 - Pipeline Execution Data of SisCEATE

SisCEATE also had an environment dedicated to training health professionals, and this training version also had the same pipeline steps as the principal repository. However, the number of versions generated for training was far less. As of August 2022, the training environment had 36 pipeline runs, with 31 successful runs, each developing a new version of the application. The success ratio of this pipeline is 86.11%. Figure 23 shows the graph of all pipeline runs over the last year.

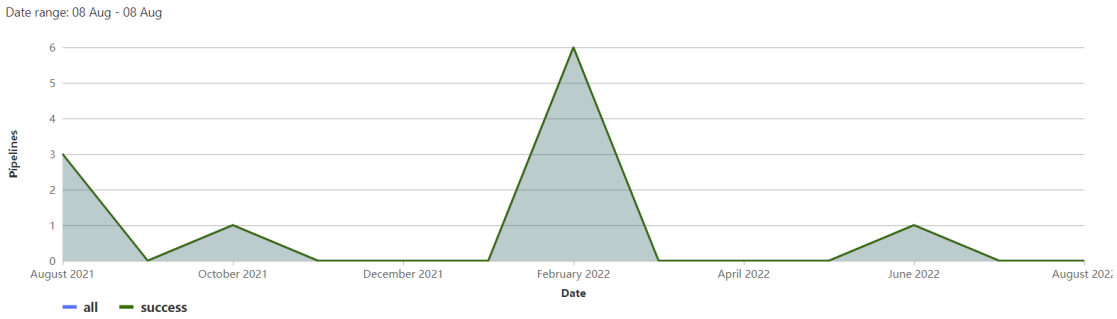

Figure 23 - Pipeline Execution Data of SisCEATE Training Environment

The third environment was their prototype, but it only had three deployments during the year. The developers used this repository to test changes in the initial stages of the system user interface. All these repositories used the same PostgreSQL instance. However, the database was not deployed to the cluster because it is difficult to manage storage on bare-metal clusters.

More recently, in the latest two months, the SisCEATE team started the development of a new application called MeuSisCEATE. This new application aims at the students in the UFRJ community. This new application already has nine successful deployments, and fourteen total pipeline runs, with a success ratio of 75%. Figure 24 presents the pipeline execution data from MeuSisCEATE in the last year.

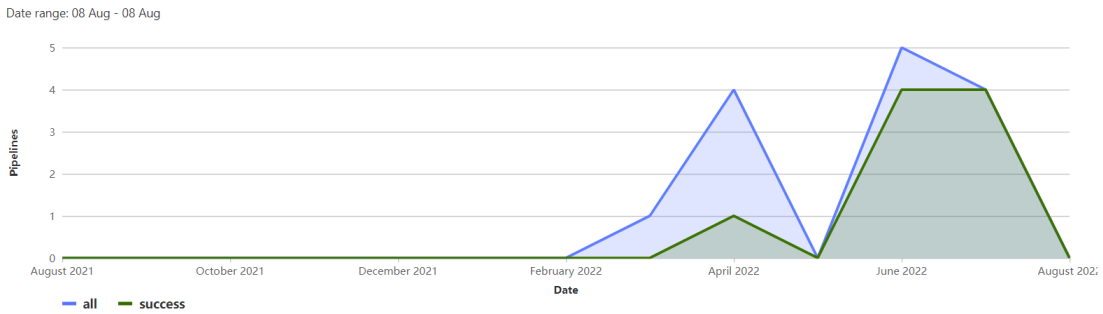

Figure 24 - Pipeline Execution Data of MeuSisCEATE Environment

Another system that used the platform for automatic deployments was the one called SAFE, a software for monitoring the usage of UFRJ installations during the pandemic. Unlike SisCEATE, a monolithic application, SAFE comprises three services, the database and one broker. Furthermore, the service architecture favors the cluster-distributed architecture since it allows the better usage of resources as K3S can schedule each of the services in a different host.

The pipeline configuration for this deployment also follows a different structure. This development team decided to use Java to build the application, and the first step in

the pipeline is to generate the application package, a JAR file. This file is then copied into a software container and shipped to the Gitlab registry. The third step in the pipeline is to build the necessary configurations in the cluster, including secrets, namespaces, and ingress for DNS resolution. Furthermore finally, the fourth and final step is to deploy all the resources regarding the application, such as the database and the application package. Figure 25 presents the stages of the pipeline for SAFE, the jobs marked with an exclamation mark are jobs with warnings or non-blocking errors.

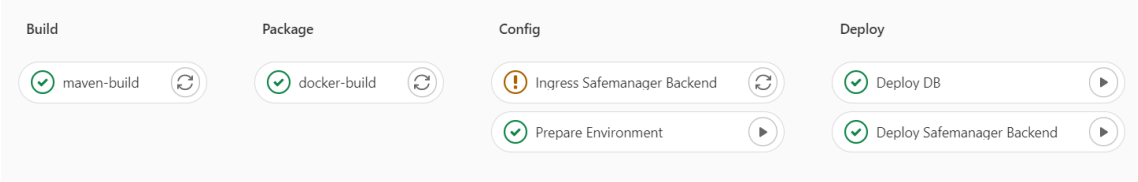

Figure 25 - Deployment Pipeline for SAFE

During the development time of SAFE, it had twenty pipeline executions and ten successful deployments. It also had two canceled deployments by the developers, causing the success ratio of this delivery pipeline to be 55.56%. Figure 26 presents the pipeline execution data for SAFE from 08/08/2021 to 08/08/2022.

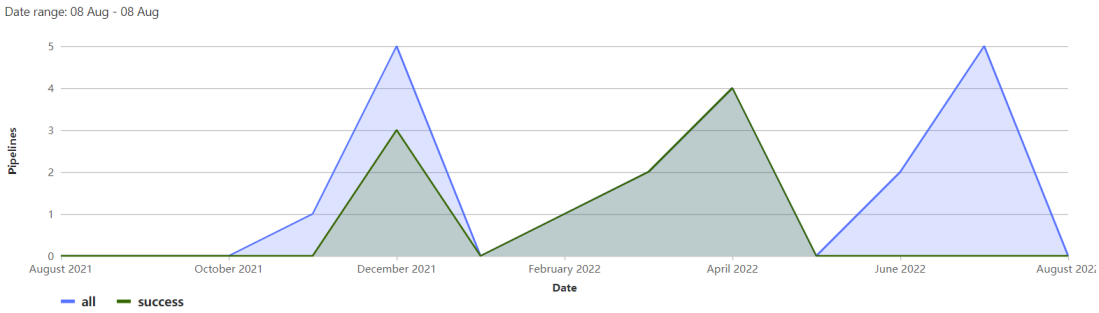

Figure 26 - Pipeline Execution Data of SAFE

During the rollout of SAFE, the MySQL database was also deployed; however, the RabbitMQ broker was not. The broker was installed separately from the services and applications deployed from Gitlab because of the deployment's complexity and because the broker did not need the frequent updates as all the other software in the development phase.

5.7. Conclusions

Based on the results and the two years of development of these applications on the proposed cluster architecture, it is possible to say that Kubernetes, specifically K3S, can be used to deploy conventional and IoT applications. However, using IoT devices

with K3S still brings up some points that need to be considered when designing a cluster for this operation.

First, while the ARM64 processor architecture is widely available and adopted, some solutions, specifically software containers, are still incompatible. In some cases, the lack of consistent solutions can slow development and trouble the conception of a release pipeline in environments that depend on these containers.

Another consideration is on applications that cannot run on multiple processor architectures, as these applications need specific configurations regarding the cluster scheduler. Without a particular configuration, the cluster can send the wrong application to the wrong device, preventing the software container from running.

Regarding the usage of brokers and other IoT services, they also present challenges regarding network configuration and strategies for self-healing nodes that may disconnect from the cluster. For example, while working with the RabbitMQ cluster, the team noticed that it is susceptible to configuration loss when all nodes go down. This scenario is prone to happen in IoT applications, as all the nodes in a network can lose internet connection or energy. In most cases, RabbitMQ needed manual intervention to return from failures caused by all nodes disconnected from the cluster.

5.8. Chapter Considerations

In this Chapter, we presented a proposed architecture for the continuous development of IoT-based systems. The architecture uses Gitlab and K3S to build, test and deploy applications in a scenario that may be used for IoT on ARM64 architecture. To test the proposed architecture, we set it up in the LENS of Coppe/UFRJ. Then, we used it to support the development of multiple applications for two years, from 2020 to 2022. In the end, the architecture proposed proved helpful for the deployments, with a few caveats, such as the high number of configurations needed to run the containers, even with the simplified setup provided by K3S. It also requires developers to be aware of the container images they are using, as they must be compatible with the nodes' processor architecture.

6 Conclusion

This chapter presents this dissertation's final considerations, highlighting the main contributions and describing the answers to the research questions. Besides, it outlines the research limitations and future works.

6.1. Final Considerations

Software containers and container orchestration tools are used constantly in modern software development projects. However, the usage of such technologies in IoT is still under discussion since there are many uncertainties if technologies already being used on conventional software can be applied in these scenarios.

In Chapter 2, we present some studies that, in different forms, apply some of these already established solutions to the context of IoT devices; however, all of them are missing a more in-depth evaluation of these technologies and devices. Furthermore, since most of the studies found in the literature are not comprehensible regarding the applications of these technologies, we developed our series of experiments to answer the first research question.

These experiments aimed to evaluate Kubernetes and the other two container orchestration tools regarding the performance in responses per second when used in a Raspberry PI 3, a popular choice of hardware for IoT projects. Based on the two experiments, Chapters 3 and 4, it is possible to answer research question one “Can conventional software container orchestration tools be used to enable continuous deployments on low-cost devices?” Yes, they can; if the hardware is compatible with cgroups and namespaces, two fundamentals for executing containers, and has enough memory for the orchestrators to run, they will be capable of achieving a container orchestration tool without significant drawbacks.

The architecture proposal and implementation at LENS helped us answer the second research question, “What are the limitations of these software container orchestration tools in the limited environment provided by low-cost devices?” As discussed in Chapter 5, by adding non-conventional hardware to the cluster, several concerns regarding CPU architecture, the building of the software images, and compatibility between components arise. In addition, most failed deployments occurred because developers were unaware of the deployment platform or the nodes could not self-heal from a scenario where most nodes could not communicate. All these perceived

difficulties would barely happen in a cloud environment for conventional software, where most of the problems that occurred in the laboratory cluster are already ironed out.

Regarding the cluster configuration in the laboratory, the cluster (and the architecture) is available and operational at k3s.cos.ufrj.br. However, other software development teams started to use the platform to support their software projects. Therefore, we decided to upgrade the cluster to Raspberry PIs 4, the new generation of Raspberry PI, with four vCores and 8 GB RAM. This update allows more applications to use the cluster and improves the performance of already existing deployments.

However, even with the described problems and limitations of the platform, we firmly believe that container orchestrators are a great solution to some of the issues presented in the IoT domain.

6.2. Contributions

Based on the objectives proposed in this Dissertation introduction, the results can be broken down into the following contributions:

- In the realization of a performance evaluation on Kubernetes on Raspberry PI 3 in synchronous and asynchronous scenarios, the results showed that it is possible to run Kubernetes using low-cost hardware commonly used in IoT projects.
- The second contribution is the performance comparison of Kubernetes, K3S, and KubeEdge as container orchestrators for the edge. The study shows no performance impacts from choosing one solution over another. However, during the setup of the environment, it became clear that some of the solutions were not yet mature enough for production usage.
- The final contribution proposes an architecture for the continuous development of IoT devices on edge. The architecture was later implemented at LENS and used by developers and students in the context of the pandemic.

6.3. Limitations

The main limitations of this dissertation are:

- One of the limitations is the generalization of the tests done during the performance evaluations. In addition, the number of tools and applications under test were few and may not represent the high number of tools and technologies used for IoT.
- The second limitation is the technical limitation regarding the hardware used to implement the bare-metal clusters. Since we need the hardware to test and use the orchestrators, the hardware available is a limiting factor.
- The third limitation regards the use of the architecture proposed; we are aware that the number of projects is not enough to validate the proposed architecture. In the future, we hope that more projects may use the architecture already established in the laboratory for more experiments and deployments.

6.4. Future Works

The perspective for future works that can be taken into consideration for the proposed architecture are:

- Adopting a test phase during the deployment pipelines on Gitlab was a step that was impossible for the current projects.
- The evaluation or more container orchestrator tools, such as Docker Swarm and many others, may be used in the context of IoT systems.
- To execute a study comparing Kubernetes components, like container network interfaces, storage solutions, and container runtimes.
- Improve the existing cluster on the LENS/ESE laboratory, allowing the laboratory to have a complete container orchestration architecture for IoT.

References

Ansible Documentation — Ansible Documentation. Disponível em: <https://docs.ansible.com/ansible/latest/index.html>. Acesso em: 2 set. 2021.

ATZORI, L.; IERA, A.; MORABITO, G. The Internet of Things: A survey. **Computer Networks**, v. 54, n. 15, p. 2787–2805, out. 2010.

BORMANN, C.; ERSUE, M.; KERÄNEN, A. **Terminology for Constrained-Node Networks**. [s.l.] RFC Editor, 2014.

BREWSTER, C. et al. IoT in Agriculture: Designing a Europe-Wide Large-Scale Pilot. **IEEE Communications Magazine**, v. 55, n. 9, p. 26–33, 2017.

C. KOLIAS et al. DDoS in the IoT: Mirai and Other Botnets. **Computer**, v. 50, n. 7, p. 80–84, 2017.

cgroups(7) - Linux manual page. Disponível em: <http://man7.org/linux/man-pages/man7/cgroups.7.html>. Acesso em: 3 mar. 2020.

Docker Documentation | Docker Documentation. Disponível em: <https://docs.docker.com/>. Acesso em: 9 jan. 2020.

Documentation :: JMeter-Plugins.org. Disponível em: <https://jmeter-plugins.org/wiki/ThroughputShapingTimer/>. Acesso em: 4 mar. 2020.

DORSEMAINE, B. et al. **Internet of Things: A Definition & Taxonomy**. 2015 9th International Conference on Next Generation Mobile Applications, Services and Technologies. **Anais**... Em: 2015 9TH INTERNATIONAL CONFERENCE ON NEXT GENERATION MOBILE APPLICATIONS, SERVICES, AND TECHNOLOGIES (NGMAST). Cambridge, United Kingdom: IEEE, set. 2015. Disponível em: <http://ieeexplore.ieee.org/document/7373221/>. Acesso em: 26 Nov. 2019

FELTER, W. et al. **An updated performance comparison of virtual machines and Linux containers**. 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). **Anais**...mar. 2015.

GEORGETA GUŞEILĂ, L.; BRATU, D.-V.; MORARU, S.-A. **DevOps Transformation for Multi-Cloud IoT Applications**. 2019 International Conference on Sensing and Instrumentation in IoT Era (ISSI). **Anais**...2019.

GitLab Documentation. Disponível em: <https://docs.gitlab.com/>. Acesso em: 7 ago. 2022.

HUANG, Y. et al. **Design and Implementation of an Edge Computing Platform Architecture Using Docker and Kubernetes for Machine Learning**. Proceedings of the 3rd International Conference on High-Performance Compilation, Computing and Communications. **Anais**...: HP3C '19.New York,

NY, USA: Association for Computing Machinery, 2019. Disponível em: <https://doi.org/10.1145/3318265.3318288>

HUMBLE, J.; FARLEY, D. **Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation**. [s.l.] Pearson Education, 2010.

K3s - 5 less than K8s. Disponível em: <https://rancher.com/docs/k3s/latest/en/>. Acesso em: 28 fev. 2020.

KATASONOV, A. et al. **Smart Semantic Middleware for the Internet of Things**. [s.l: s.n.]. v. 8

KAWABATA, H. et al. Robust Relay Selection for Large-Scale Energy-Harvesting IoT Networks. **IEEE Internet of Things Journal**, v. 4, n. 2, p. 384–392, abr. 2017.

Kubernetes. Disponível em: <https://kubernetes.io/pt/docs/home/>. Acesso em: 20 jan. 2020.

LWAKATARE, L. E. et al. **Towards DevOps in the Embedded Systems Domain: Why is It So Hard?** 2016 49th Hawaii International Conference on System Sciences (HICSS). **Anais**... Em: 2016 49TH HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES (HICSS). Koloa, HI: IEEE, jan. 2016. Disponível em: <http://ieeexplore.ieee.org/document/7427859/>. Acesso em: 26 nov. 2019

MEKKI, K. et al. A comparative study of LPWAN technologies for large-scale IoT deployment. **ICT Express**, v. 5, n. 1, p. 1–7, mar. 2019.

MERKEL, D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. p. 5, [s.d.].

MOORE, J. et al. **DevOps for the Urban IoT**. Proceedings of the Second International Conference on IoT in Urban Space - Urb-IoT '16. **Anais**... Em: THE SECOND INTERNATIONAL CONFERENCE. Tokyo, Japan: ACM Press, 2016. Disponível em: <http://dl.acm.org/citation.cfm?doid=2962735.2962747>. Acesso em: 26 Nov. 2019

MORABITO, R. **A performance evaluation of container technologies on Internet of Things devices**. 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). **Anais**... Em: IEEE INFOCOM 2016 - IEEE CONFERENCE ON COMPUTER COMMUNICATIONS WORKSHOPS (INFOCOM WKSHPS). San Francisco, CA, USA: IEEE, abr. 2016. Disponível em: <http://ieeexplore.ieee.org/document/7562228/>. Acesso em: 26 Nov. 2019

MORABITO, R. Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation. **IEEE Access**, v. 5, p. 8835–8850, 2017.

MORABITO, R. et al. Evaluating Performance of Containerized IoT Services for Clustered Devices at the Network Edge. **IEEE Internet of Things Journal**, v. 4, n. 4, p. 1019–1030, ago. 2017.

MOTTA, R. C.; DE OLIVEIRA, K. M.; TRAVASSOS, G. H. On Challenges in Engineering IoT Software Systems. **Journal of Software Engineering Research and Development**, v. 7, p. 5:1-5:20, set. 2019.

MOTTA, R.; SILVA, V.; TRAVASSOS, G. Towards a more in-depth understanding of the IoT Paradigm and its challenges. **Journal of Software Engineering Research and Development**, v. 7, p. 3:1--3:16, 2019.

Multi-architecture plan for Kubernetes · Issue #38067 · kubernetes/kubernetes. Disponível em: <https://github.com/kubernetes/kubernetes/issues/38067>. Acesso em: 3 mar. 2020.

namespaces(7) - Linux manual page. Disponível em: <http://man7.org/linux/man-pages/man7/namespaces.7.html>. Acesso em: 4 mar. 2020.

NIELSEN, J. Chapter 5 - Usability Heuristics. Em: NIELSEN, J. (Ed.). **Usability Engineering**. San Diego: Morgan Kaufmann, 1993. p. 115–163.

RAZZAQUE, M. A. et al. Middleware for Internet of Things: A Survey. **IEEE Internet of Things Journal**, v. 3, n. 1, p. 70–95, fev. 2016.

RUFINO, J. et al. **Orchestration of containerized microservices for IIoT using Docker**. 2017 IEEE International Conference on Industrial Technology (ICIT). **Anais**... Em: 2017 IEEE INTERNATIONAL CONFERENCE ON INDUSTRIAL TECHNOLOGY (ICIT). Toronto, ON: IEEE, Mar. 2017. Disponível em: <http://ieeexplore.ieee.org/document/7915594/>. Acesso em: 26 Nov. 2019

SZILAGYI, I.; WIRA, P. **Ontologies and Semantic Web for the Internet of Things - a survey**. [s.l: s.n.].

TOMA, I.; SIMPERL, E.; HENCH, G. A joint roadmap for semantic technologies and the Internet of Things. 1 jan. 2009.

What is KubeEdge — KubeEdge Documentation 0.1 documentation. Disponível em: <http://docs.kubeedge.io/en/latest/modules/kubeedge.html#advantages>. Acesso em: 28 fev. 2020.

XINGMEI, X.; JING, Z.; HE, W. **Research on the basic characteristics, the key technologies, the network architecture, and security problems of the Internet of things**. Proceedings of 2013 3rd International Conference on Computer Science and Network Technology. **Anais**... Em: 2013 3RD INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND NETWORK TECHNOLOGY (ICCSNT). Dalian, China: IEEE, out. 2013. Disponível em:

<http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6967233>. Acesso em: 26 Nov. 2019

ZAMBONELLI, F. Key Abstractions for IoT-Oriented Software Engineering. **IEEE Software**, v. 34, n. 1, p. 38–45, 2017.

Appendix A – Operation Manual for K3S and the Services Installed on the Cluster

Management and Configuration of the K3S Cluster

Since we decided to keep K3S as the cluster for the LENS/ESE lab, it is fundamental to relay instructions on how to operate the cluster. First, K3S is distributed as a single binary file. This way, the installation and update of the cluster and its components can be easily done by downloading and replacing the files in the respective folders. The K3S binary is in the path **/usr/local/bin/k3s.** Usually, K3S can be updated using an automated script such as **curl -fsL https://get.k3s.io. H**owever, updating this way is not recommended since the wrong parameters can overwrite the already existing configuration of the cluster. The second way to update, install or update K3S is to replace the binary file on the corresponding path, which is the best way in the current cluster configuration.

Currently, the K3S cluster installed uses the same PostgreSQL database on the Gitlab Server VM on PESC. The current configuration for the K3S service file specifies the connection string to the database and the encryption token to join new nodes to the cluster. To join new nodes, the only command needed is **curl -fsL https://get.k3s.io | K3S_URL=https://SERVER_URL:6443 K3S_TOKEN=ENCRYPTION_TOKEN sh –** this command will allow the new host to join the cluster, and no other tools are necessary to run the cluster services.

After the installation and setup of the master and worker nodes, the binary **kubectl** is available for the root user to allow the configuration and management of the cluster, but it can only be used in the master node. The K3S installation comes, by default, with the following components:

- **Kubernetes Metrics Server**: Collects container resource metrics for Kubernetes.
- **Flannel**: Container network fabric for Kubernetes. It provides a layer 3 IPv4 network between the nodes in a cluster.
- **Traefik**: A open-source edge router for Kubernetes, it is the main Ingress component of the cluster, receiving incoming traffic in a specified DNS and routing them to the appropriate component.

- **CoreDNS**: It`s a DNS Server and Service Discovery tool that allows containers to communicate with each other based on internal service names on Kubernetes.

Managing K3S Nodes and Services

In K3S, the management of nodes is no different from a regular Kubernetes cluster. To list the nodes joined in a cluster, the user can use the **kubectl get nodes. T**he command can be expanded using **kubectl get nodes -o wide** to show each node IP, hostname, operating system version, and Kubernetes version. For a user to join new nodes in the cluster, the user must use the command described in the first section of this appendix. It is recommended to first delete the resources in the node to completely remove a node from the cluster. To do that, the user must execute the following actions in sequence:

1. **kubectl cordon <NODE_NAME>:** This will mark the node unschedulable.
2. **kubectl drain <NODE_NAME>:** This command will remove all the Kubernetes resources from the node.
3. **kubectl delete <NODE_NAME>:** This will completely remove the node from Kubernetes.

K3S and Flannel will set up the network configuration for all nodes alone during the cluster setup. Therefore, users do not need to manually delete or change iptables configuration or delete the network interfaces from nodes. Although in K3S, the containers related to Flannel are not visible to the user, the networking configuration must be changed using the K3S service file as arguments to the K3S binary. After changing the configuration, the user must trigger a daemon-reload of the Linux service stack and restart the K3S main service. The commands to execute these tasks are **systemctl daemon-reload** and **systemctl k3s restart**.

Applying Configuration to the Cluster

Kubernetes is a state machine that uses a declarative system, which means that to achieve a desired state in the cluster, the user must provide a configuration file to tell the cluster the state he wants to achieve. Kubernetes is compatible with two different configuration files, YAML and JSON, in which, with a set of declarations, the Kubernetes objects are declared. The most common Kubernetes objects are listed below:

- **Namespaces**: Namespaces are virtual clusters used to logically separate the resources in the cluster.
- **Pods**: These are the smallest unit of deployment in a Kubernetes cluster. They have a unique IP address and can communicate with the rest of the cluster and host one or more containers.
- **Deployments**: These are controller objects; they instruct Kubernetes on how to manage the pods in an application. On the deployment configuration file, the user can set the number of pod replicas, roll out updated code or perform a rollback of older code versions.
- **Services**: Services provide a way to expose applications running on pods to the cluster or externally. The main purpose of this object is to map the pod IP to the virtual IP created when the service is started. This behavior allows the pods to be recreated in the event of a failure without the user having to manually intervene to update the IP on all connected devices or applications.
- **Volumes**: These objects provide storage to pods, usually persistent or ephemeral volumes.
- **DaemonSets**: DaemonSets are special controller objects that ensure that a pod replica runs on all cluster nodes. The Kubernetes scheduler ignores pods created by DaemonSets, so the pods can run on the nodes as long as the node exists.
- **StatefulSets**: StatefulSets is a special controller that initiates pods with stateful apps. The pods created by StatefulSets have their unique network ID, persistent storage and ordered deployment, scaling and rollout that persist even when the pods are restarted.
- **ConfigMaps**: ConfigMaps are key-value pairs used to store configuration data that can later be used and referenced by applications and pods. It allows the decoupling configuration files and the application being run on Kubernetes.
- **Jobs**: While Deployment Controllers and the other have the task of permanently maintaining several applications and pods running, keeping their desired state, the Job Controller executes finite tasks and then terminates the associated pod. These can run scripts necessary for a specific application, create tables, or run particular queries.

- **Secrets**: A Secret is an object with a small amount of sensitive data, such as a password, token or key. These, like the ConfigMap, can be referenced by a pod or application, allowing the user and developers to not include confidential data in the application.
- **Ingress**: Ingress is an API object type that allows external access to the services in the cluster using HTTP or HTTPS.

The command used to apply any configuration to any object in Kubernetes is **kubectl apply -f** followed by the complete file path and name of the configuration file. If the user wants to delete any configuration from Kubernetes, the command is **kubectl delete -f,** followed by the configuration file. It is also possible to edit configurations already deployed to Kubernetes, using the command **kubectl edit** followed by the name of the configuration to be edited. These commands can be used to edit any object in the Kubernetes cluster.