NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Decision Support Systems Laboratory

Development of an IoT data processing system using distributed technologies based on Kubernetes

DIPLOMA THESIS

Nikolas T. Bellos

SUPERVISOR

Vangelis Marinakis
Assistant Professor

Athens, February 2024

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Ηλεκτρικών Βιομηχανικών Διατάξεων & Συστημάτων Αποφάσεων
Εργαστήριο Συστημάτων Αποφάσεων και Διοίκησης

Ανάπτυξη συστήματος επεξεργασίας IoT δεδομένων με χρήση κατανεμημένων τεχνολογιών βασισμένο σε Kubernetes

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Νικόλας Θ. Μπέλλος

ΕΠΙΒΛΕΠΩΝ

Μαρινάκης Ευάγγελος
Επίκουρος Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις 29 Φεβρουαρίου 2024.

.....	.....	.....
Μαρινάκης Ευάγγελος Επίκουρος Καθηγητής	Δούκας Χάρης Καθηγητής	Ψαρράς Ιωάννης Καθηγητής

Αθήνα, Φεβρουάριος 2024.

Abstract

The Internet of Things (IoT) era is here, with the number of internet-connected devices expected to hit over 75.4 billion by 2025. These devices create lots of data that help us understand and control our environment from afar. Traditionally, the systems managing this data have been closed-source and outdated by the time they're widely available. Now, cloud-native technologies and virtualization make it easier and more efficient to manage IoT systems. Kubernetes, a key tool, helps make building and maintaining IoT infrastructures simpler and less manpower-intensive.

This thesis explores the design and implementation of an IoT system using distributed technologies deployed on Kubernetes, an open-source framework for automating container application management, deployment, and scaling. It discusses the need for efficient processing and management of large data volumes generated by numerous IoT devices and how implementation through Kubernetes offers a flexible and dynamically scalable solution. Additionally, it examines technologies like MQTT for efficient message transfer and Apache Kafka for real-time data processing and storage. It also analyzes the importance of using distributed systems to improve performance, scalability, and resilience compared to centralized systems, as well as the challenges encountered in managing distributed data and achieving consensus in such environments. Finally, it presents suggestions for future work, including automated scaling, monitoring metrics, continuous integration/delivery (CI/CD), and the extensive implementation of security techniques.

Keywords: Kubernetes, IoT, Distributed systems, Microservices, Systems Design, Data-driven microservices, Kafka, MQTT, Smart Home, Cloud computing, Containers, Energy Consumption.

Περίληψη

Το Internet of Things (IoT) είναι εδώ, με τον αριθμό των συνδεδεμένων στο διαδίκτυο συσκευών να αναμένεται να ξεπεράσει τα 75,4 δισεκατομμύρια έως το 2025. Αυτές οι συσκευές δημιουργούν πολλά δεδομένα που μας βοηθούν να κατανοήσουμε και να ελέγξουμε το περιβάλλον μας από μακριά. Παραδοσιακά, τα συστήματα που διαχειρίζονται αυτά τα δεδομένα ήταν κλειστού κώδικα και ξεπερασμένα κατά τη στιγμή που έγιναν ευρέως διαθέσιμα. Πλέον, οι τεχνολογίες cloud-native και containerization καθιστούν πιο εύκολη και αποτελεσματική τη διαχείριση των συστημάτων IoT. Το Κυβερνετες, ένα βασικό εργαλείο, βοηθά στην απλούστευση της κατασκευής και συντήρησης των υποδομών IoT κάνοντας τη διαδικασία λιγότερο εντατική σε εργατική δύναμη.

Αυτή η διατριβή εξερευνά τον σχεδιασμό και την υλοποίηση ενός συστήματος IoT χρησιμοποιώντας διανεμημένες τεχνολογίες που αναπτύσσονται στο Kubernetes, ένα ανοιχτού κώδικα πλαίσιο για την αυτοματοποίηση της διαχείρισης, της ανάπτυξης και της κλιμάκωσης εφαρμογών σε containers. Συζητά την ανάγκη για αποτελεσματική επεξεργασία και διαχείριση μεγάλων όγκων δεδομένων που παράγονται από πολυάριθμες συσκευές IoT και πώς η υλοποίηση μέσω του Kubernetes προσφέρει μια ευέλικτη και δυναμικά κλιμακούμενη λύση. Επιπλέον, εξετάζει τεχνολογίες όπως το MQTT για αποτελεσματική μεταφορά μηνυμάτων και το Apache Kafka για την επεξεργασία και αποθήκευση δεδομένων σε πραγματικό χρόνο. Αναλύει επίσης τη σημασία της χρήσης διανεμημένων συστημάτων για τη βελτίωση της απόδοσης, της κλιμάκωσης και της ανθεκτικότητας σε σύγκριση με τα κεντρικοποιημένα συστήματα, καθώς και τις προκλήσεις που συναντώνται στη διαχείριση διανεμημένων δεδομένων και την επίτευξη συναίνεσης σε τέτοια περιβάλλοντα. Τέλος, παρουσιάζει προτάσεις για μελλοντική εργασία, συμπεριλαμβανομένης της αυτοματοποιημένης κλιμάκωσης (autoscaling), του monitoring μέσω metrics, του continuous integration/delivery (CI/CD) και της εκτεταμένης υλοποίησης τεχνικών ασφαλείας.

Preface

This thesis summarizes my work at the EPU lab located in NTUA during my last semester of studies. It marks the culmination of my work during the past 5 years as an Electrical Engineering & Computer Engineering student. Although the subjects I completed concern a broader engineering spectrum, my specialization concerns software engineering, computer systems and Kubernetes.

To follow this path, there were some people that played a key role and whom I would like to thank. First of all, I would like to thank Vangellis Marinakis for trusting me and offering me a place at the lab, providing me with the opportunity to work on this project. I am also grateful for Elissaios Sarmas and Vasilis Michalakopoulos who provided me with all the needed guidance and assistance when I needed it. The two people that I worked closer with for this project were Fillipos Serepas and John Papias, whom I would like to thank especially for the help, the calls and for trusting me to 'break' things. The story on how I ended up learning about Kubernetes is longer than that and for obtaining all the basic skills and confidence in this field I would like to thank the people of Arrikto and Vaggelis Koukis.

Last but not least, I would like to express how appreciative I feel about studying in this university and along many talented people that I now call friends. I saved thanking my family for last, for the love they always provided and the shaping of my character. I dedicate this work to them.

Nikolas Bellos

NTUA, Athens
29nd February 2024

Contents

Nomenclature

AI Artificial Intelligence

CI/CD Continuous Integration / Continuous Development

CPU Central Processing Unit

DNS Domain Name System

EHS Energy Home System

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IoT Internet of Things

ML Machine Learning

MQTT Message Queuing Telemetry Transport

SQL Structured Query Language

SSL Secure Sockets Layer

TCP Transmission Control Protocol

URL Uniform Resource Locator

VM Virtual Machine

WAL Write Ahead Log

YAML Yet Another Markup Language

Chapter 1

Introduction

The era of the Internet of Things (IoT) has begun and it will be relevant for the upcoming years. Billions of devices are already connected to the Internet with their number being projected to surpass 75.4 billion by 2025. This surge in the number of IoT devices results consequently in greater volumes of data being generated and processed in order to gain insightful information about the world around us and control it remotely. This highlights the need for message delivering and processing solutions, around all different IoT system types.

So far, there have been developed many solutions provided by third party providers or provided through a cloud provider platform. The state-of-the-art software and architecture, therefore, for these systems is mainly closed-sourced and when it becomes available to the general public after some years, it lags behind the current state of technology and the actual demands of the real-world infrastructure. At the same period, cloud native technologies have become more and more democratized and tools that can run in a decentralized and distributed manner are more reliable and production ready than ever.

Virtualization technology has also played a key role in deploying these systems, as a single machine can now be used to host multiple smaller host kernels on top of which containers can run. Resource allocation becomes immediately more efficient and the initialization time of individual applications decreases significantly. The amount of people needed to deploy and maintain such a system also decreases and with the help of Kubernetes, the building of custom IoT infrastructure becomes more accessible and more easily maintainable. The question that remains to be answered is whether the technologies that exist today can be performant and reliable enough to replace black-box solutions provided by third party providers.

The processing and collection of these data is not the only driving force

for this work. The topics of energy and artificial intelligence are also very ubiquitous and relevant for the field of computer systems and information processing. To lean towards a more energy efficient future, we must be able to monitor the energy consumption of various power hungry devices and take the appropriate actions. This can be achieved by providing the device users the appropriate data, by advising them on their usage habits, by distributing the load in a more efficient way across the power grid, or by providing the data to authorities, which can later pass relevant laws or take immediate action on problematic occasions. Machine Learning techniques can also assist in this matter, but for them to actually work, massive data of good quality are needed to ensure the high accuracy of the developed ML models.

The thesis lays the groundwork for a system that utilizes open-source technologies in order to handle and process various IoT data. It also analyses the suitability of Kubernetes as a framework that can stand beside commercial cloud infrastructure and whether it can offer performance and reliability as much as it offers control.

1.1 Thesis structure

Before getting into the more detailed sections of this work, we should first try to provide a brief description of each of its 6 distinct sections, in order to get a more holistic view of it.

In the first section of this work, we present the problem statement and motivation, as well as relevant questions that are worth to be answered.

The second section deals with all the background information that one needs to be familiar with, in order to understand the theoretical aspects of this work from the ground up. Also, a current state of the work in this field is given, along with more general diagrams to gradually introduce the reader to the next section.

The third one is about the design and architecture of the system. We present the different software components that we used to create the central message processing pipeline and how they are connected with each other to ensure a truly distributed system. When refering to the system, we talk about an infrastructure about IoT message processing based on Kubernetes.

In the fourth section we describe the deployment of the system. We describe the open-source software tools that we used and how we deployed them in Kubernetes. An almost step-by-step configuration process is provided, with all the details of the infrastructure and the helper links one can use to setup a similar system in Kubernetes.

The fifth section consists of our findings while analyzing the deployed

system. We evaluate its capacity and scalability based on performance testing metrics. We explore the message throughput and resource needs for our 3-Node Kubernetes cluster.

In the sixth and final section, we summarize the architectural decisions and the performance outcomes of the system and propose future extensions for the system.

1.2 Motivation

This work describes the decision-making process and the tradeoffs for designing and deploying an IoT infrastructure using only open-source and distributed software tools that can be deployed in Kubernetes, an open-source container orchestration framework. The need for such a system stems from the development of EHS, which stands for Energy Home System. The idea behind it is to create the necessary infrastructure, in order to collect, process and store data from energy meter sensors.

These data have the potential to assist not only the user in keeping an eye on his energy usage, but also a central authority in gaining knowledge about household energy consumption patterns and training machine learning models to allocate resources for home appliances more effectively.

Furthermore, the majority of such systems that gather and process messages in very large scale are closed sourced, both the design and the source code. Nowdays, tools and resources are provided as a service in the cloud, which is an adequate and easy solution for most cases. However, there is a need to open-source tools and methodologies regarding the cutting edge of IoT data processing systems, as it is a field that experiences exponential growth at the moment. These tools have the potential to democratize the development of distributed systems based on Kubernetes and provide the laying ground for labs ans smaller companies to experiment with them, thus resulting in more reapid growth in the field of IoT systems.

More specific applications of the aforementioned system are presented below:

Energy Autonomy

With the data collected and by applying AI/ML algorithms, accurate predictions of energy production and consumption can be achieved, even in scenarios with suboptimal data quality. This capability enables users to gain insights into their energy ecosystem, including precise information on energy generation and consumption. Therefore, users can now compare their en-

ergy consumption with other similar domestic installations, allowing them to make informed decisions on load-shifting strategies to optimize their energy consumption by strategically aligning it with the solar energy production, thereby extending the hours of energy autonomy during the day.

Demand Response

Through a single measurement capturing total consumption and employing cutting-edge algorithms for "energy disaggregation", users gain the ability to discern the energy consumption of individual devices within their homes. This understanding empowers users to identify and address energy-intensive devices, whether through more efficient usage or by considering replacements. From the utility company's standpoint, this technology offers real-time visibility into the consumption patterns of a multitude of households. This perspective allows utility companies to pinpoint the most flexible areas in terms of demand, enabling strategic implementation of Demand Response initiatives.

Remote Control

By integrating swiftly deployable equipment, dedicated monitoring, and evaluation tools offer a detailed breakdown of consumption per device or system (e.g., air conditioners, heaters, water heaters). This setup not only facilitates remote monitoring but also enables remote control. Leveraging AI/ML algorithms, these systems are trainable to provide personalized suggestions tailored to each user's energy behavior and the installed appliances. Users can transition towards a hands-off approach, with minimal initial input such as setting basic conditions (temperature, hours away from home). The Energy Management System (EMS) takes center stage, continuously optimizing energy consumption without the need for constant user intervention. This transition from active intervention to automated optimization not only simplifies the user experience but also ensures sustained efficiency through ongoing optimization by the EMS.

1.3 Research Questions

RQ1. Is it possible to use open-source production level software that can run in a distributed manner for all components of such a system ?

Currently, the state of the art systems for message processing of large IoT clusters are proprietary and provided as a service from large cloud providers. Such systems that can tolerate high amounts of traffic are dependent on big scale infrastructure, sophisticated software and well though out architectures. It is interesting, for that reason, to investigate ways with which one can replicate these systems with home owned hardware, to know the requirements for such a smaller scale system and if it can replace the current cloud-based solutions offered as a third party service.

RQ2. How does the system perform while using the minimum amount of resources for its distributed components ?

The resources we usually have available while researching these kind of experimental technologies are limited to a few servers. Thus, we should be able to determine at what extent we can still have a distributed, horizontally scalable system, while tolerating at the same time an adequate enough amount of message load from IoT devices. By understanding how many computer resources are needeed to handle a few hundred messages per second, we can then proceed to design how the system should scale in order to handle multiples of these loads.

The current thesis is not concerned with a high message throughput tolerance system, but rather lay the foundations for a Kubernetes-based system which includes only truly-distributed components that can scale horizontally in the future, one by one.

RQ3. What are the practical limitations of such a system ?

Because this is an experimental technology and architecture, it is expected to pose many downsides compared to the state-of-the-art technologies provided by cloud providers and other third party services. By acknowledging what these limitations are, we can focus to mitigating them with a more targeted approach in the future, as well as contribute to the open-source community with tools and propositions to augment the existing software tools and architectures.

Chapter 2

Background & Related Work

2.1 Internet of Things (IoT)

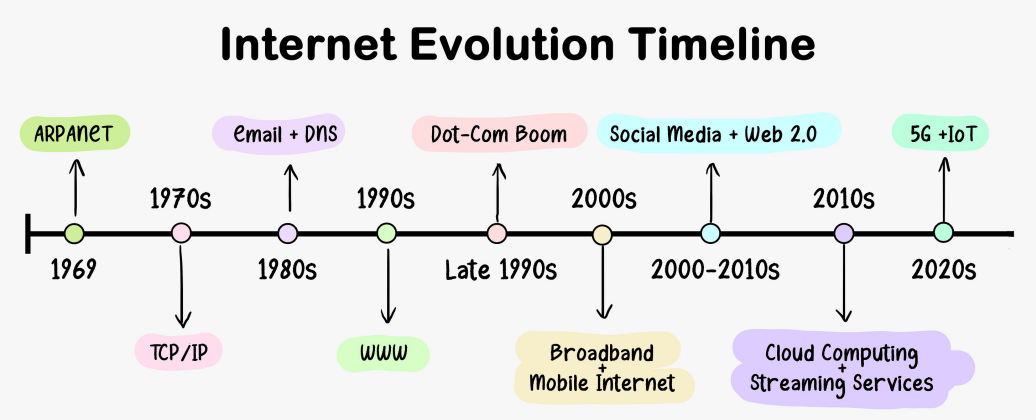

Figure 2.1: Timeline of internet evolution [1]

The history of the internet is not very long. The first network, called ARPANET, was developed by DARPA and is just 55 years old. The recent history of the internet as we know it, however, is not older than 20 years. Mobile internet was introduced around 20 years ago with the era of cloud computing beginning not earlier than the early 2010s. Until the end of the previous decade, the need for multiple wireless devices was not immense. In each household, there was a necessity for a few portable electronic devices and maybe some more sophisticated smart home devices like a wireless light switch. In the past few years, more and more problems have started to arise, which align with the progress of technology. More specifically, due to climate change, cities must learn to be more energy efficient, energy grids to

be smarter, and more data needs to be collected to make better decisions. At the same time, internet bandwidth gets bigger and bigger and data transfers get cheaper and cheaper. This can only lead to an increase in smart remote devices, which will be able to collect data across households, cities, crop fields, and many more. It is estimated that more than 75 billion smart devices will be connected to the internet by the end of 2025. We anticipate that there will be more than 9 smart devices per person in the following years [2].

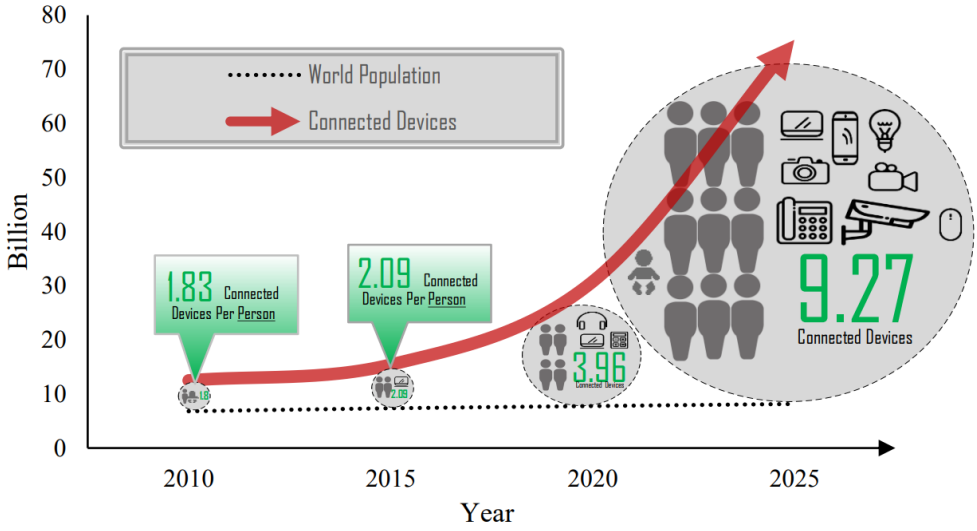

Figure 2.2: Estimated Number of Connected Devices Per Person [2]

When it comes to the IoT market, it is also experiencing significant growth with a forecast of rising to 33 billion dollars by 2027 [3]. In the smart home industry alone we expect to have 350 million devices by 2024. IoT and AI are also going to be tightly coupled in the following years, with sensors providing all the necessary data for ML models to be trained. AI in general is going to enable smart decision-making, optimize efficiencies, and provide predictive insights, making IoT devices more intelligent and efficient [4].

However, the rapid growth of these interconnected devices raises challenges. Challenges such as high data traffic, energy efficiency, and highly scalable message processing infrastructure. Thus, systems like EHS must be able to scale well, while being highly available and energy efficient at the same time. When it comes to the communication protocol with the IoT devices, MQTT is the most well-adapted one that employs the publish-subscribe architecture and is the one we will choose to set up our system as it also fits with our use case scenario where all devices are within a house environment.

2.2 MQTT Protocol

MQTT is an open source, application layer, publish-subscribe messaging protocol designed for lightweight M2M (Machine to Machine) and IoT communications [2].

MQTT (Message Queueing Telemetry Transport), initially released in 1999 by IBM employee Andy Standford-Clark, has evolved into a core component of various IoT solutions. Designed for constrained devices and networks with low bandwidth, high latency, or unreliability, MQTT minimizes network bandwidth and hardware requirements while ensuring reliability and some degree of assurance of data delivery [5]. It is typically used for transferring small messages, a few bytes in size.

2.2.1 Message Topics

In a publish-subscribe architecture, each MQTT message should have a UTF-8 string, known as "topic". Topics in MQTT are strings used by the broker to categorize messages for each device connection, often structured hierarchically for real-case scenarios (e.g. myHome/livingRoom/temperature). More details about the MQTT Brokers and the handling of those messages will be presented later.

2.2.2 Message Structure

MQTT messages have a structure consisting of a fixed header, an optional variable header, and an optional payload. The fixed header is 2 to 5 bytes, and it is variable because it represents the remaining length of the message.

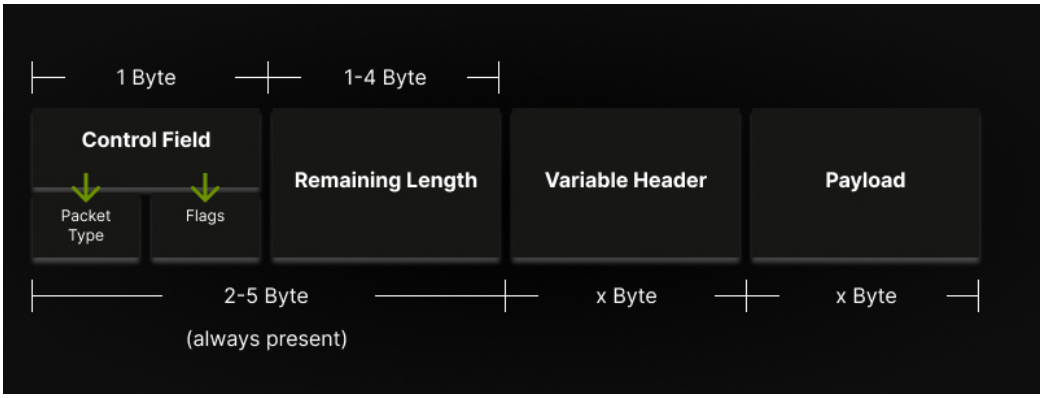

Figure 2.3: MQTT Message Structure

2.3 Containers

Containers [6] is the most straightforward way to deploy apps in the cloud. They are more lightweight than VMs, making them more energy efficient but also faster to get deployed, not to mention that they have a smaller startup time.

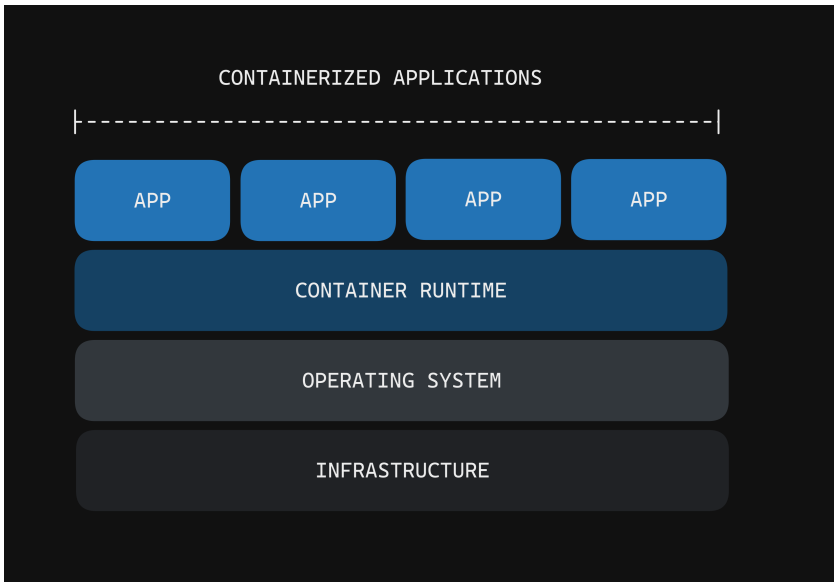

Figure 2.4: Containerized applications diagram

Therefore, they replaced relatively quickly the old way of setting up a VM and then deploying the app there by hand. This solved the problem of a more lightweight and faster deployment for a single application, which in many cases is adequate, mainly speaking for small organizations and systems with few clients and few applications to deploy. However, most applications nowadays need to operate at scale, meaning that they need to be able to scale dynamically, load-balance the traffic, get monitored, heal with minimal human intervention, and operate in a distributed way across multiple machines. All these problems are solved with a container orchestration system.

2.4 Container Orchestration Systems

A container orchestration system enables the deployment and distribution of containers across multiple physical machines, by creating virtual networks across containers and managing them from a single interface. Additionally, some of these containers running can be used as automated operators that

upscale or downscale other containers, based on their usage, or load-balance traffic that is incoming to the cluster and the different containers [7].

Kubernetes is the most popular among the existing frameworks and it is the one that we will use to deploy our proposed system.

2.5 Kubernetes

Kubernetes, often referred to as K8s, is a container-management system that assists application developers in easily deploying, scaling, and maintaining cloud-native applications.

Kubernetes is open-source, being initially developed by Google in 2014. It is derived from its internal container management software, called Borg [8]. As the popularity of containers and the cloud infrastructure arose among developers, Google got motivated to develop and open-source the well-known framework. Among its main uses are, deployment, updating, monitoring, scaling, managing, and distributed execution of containerized applications.

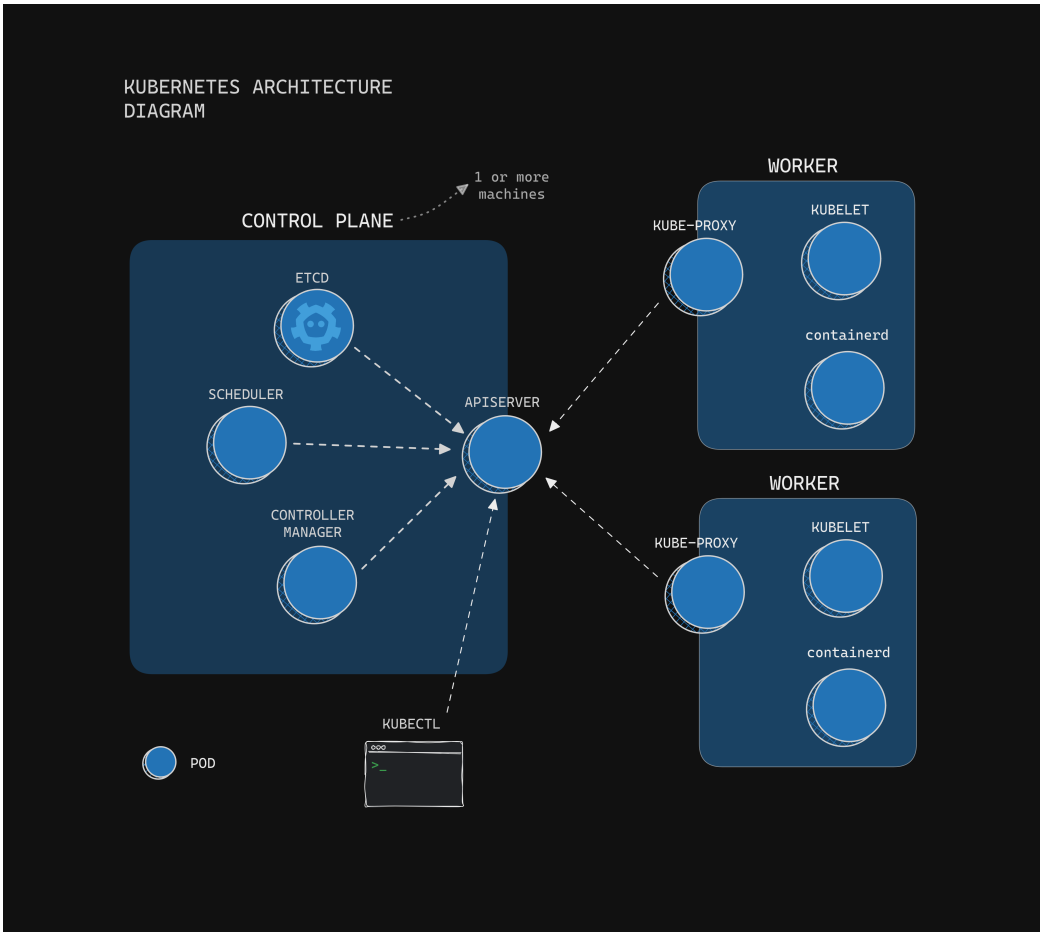

Figure 2.5: Architecture of Kubernetes

2.5.1 Components of Kubernetes

- **Control Plane & Workers**

  Kubernetes follows a master-slave architecture, where each machine is called a Node.

  The master component is represented by a cluster of Master Nodes (in bigger clusters multiple master nodes are deployed to avoid single points of failure and have better load balance across them). These Nodes are responsible for managing the CRDs and controlling the orchestration of containers through a scheduler, a controller, and an apiserver.

  The Worker Nodes are the machines that do all the heavy lifting by running the pods.

- **Etcd**

  Etcd is widely known as the go-to key-value store for Kubernetes. It can run in a distributed manner making it highly available, which is important for clusters with high amounts of Nodes and load.

  Its primary use is for storing all configuration data of the K8s cluster, whether that is the metadata of each Node or information about the Pod scheduling and status.

- **Kube-apiserver**

  All functions in K8s are exposed through a REST API. This API endpoint is referred to as apiserver, it runs on the control plane nodes and is designed to scale horizontally to be able to load balance traffic and serve all the incoming requests to the cluster, either from inside or outside of it.

- **Kube-scheduler**

  Every time a new Pod is created, the scheduler needs to be the one that chooses the Node for the Pod to run on. What it does is watch if there are Pods that do not have an assigned Node in their metadata and then choose one with the greedy approach of choosing the Node that has at least as many resources as the Pod requests (meaning CPU, memory, and storage).

- **Kube-controller-manager**

  It is the Pod that runs controller processes. When we talk about controllers, we refer to control loops that provision and act upon a change of an object, like the creation of a Pod.

  There are multiple default controllers compiled into the controller manager each taking its name depending on its function. Node controller provisions when a Node goes down for example and Deployment controller handles the actions that need to be done for a Deployment object to work properly. [IoT device management using Kubernetes → Control plane components]

- **Kube-proxy**

  It acts like most network proxies, exclusively for each of the Nodes in the cluster.
  It allows incoming and outgoing traffic to and from the Pod and maintains network rules to properly route traffic from other Pods or from

outside the cluster. It is responsible for implementing, through iptables, the Service concept by mapping containers to Services and utilizing load-balancing mechanisms to distribute traffic among them. The Service concept is going to be explained later [9].

- **Kubelet**

  In every Node a Pod has to be running, whose purpose is to communicate with the control plane and supervise the lifecycle of the Pods running on the same Node, either that has to be their creation, their failure, or health status. More specifically, it communicates with the master nodes to receive instructions about the Pods and it updates the etcd accordingly for changes it performs.

- **Containerd**

  At the heart of every Kubernetes node, there is a container runtime, a daemon responsible for running the containers inside the Pods. In our setup, we use containerd, which was, and still is, one of the main components of the Docker daemon. Containerd is a representative of the implementation of K8s CRI.
  A CRI (Container Runtime Interface) is provided for the communication with a container runtime through a high-level interface. The CRI is the main protocol for the communication between the containerd and the Kubelet.

  By the time of writing, at least four fully CRI-compatible container runtimes exist: containerd, CRI-O, Docker Engine, and Mirantis Container Runtime.

- **Kubectl**

  There are several ways to interact with the API server. One of the popular options covering the most functionality is the CLI tool Kubectl. Kubectl is the command-line tool for users to interact with the Kubernetes cluster that comes shipped along the K8s setup.

2.5.2 CRD (Custom Resource Definition)

Kubernetes uses 'objects' it calls CRDs to give meaning to the different entities that it can understand [10].

There are predefined CRDs in Kubernetes (like Deployment), but one can also create his own and apply them through manifests [10], in the format of yaml files **ansibleYAMLSyntax**. Kubernetes can understand CRDs deployed in the format of a yaml file.

Each CRD is defined with a schema. One that wants to deploy a resource in Kubernetes must follow the structure of that schema and provide all the necessary attributes that are described in it. The structure of a CRD can be separated into two parts. The universal information is described below the 'metadata' section while the CRD specific is described below the 'spec' section of the yaml file.

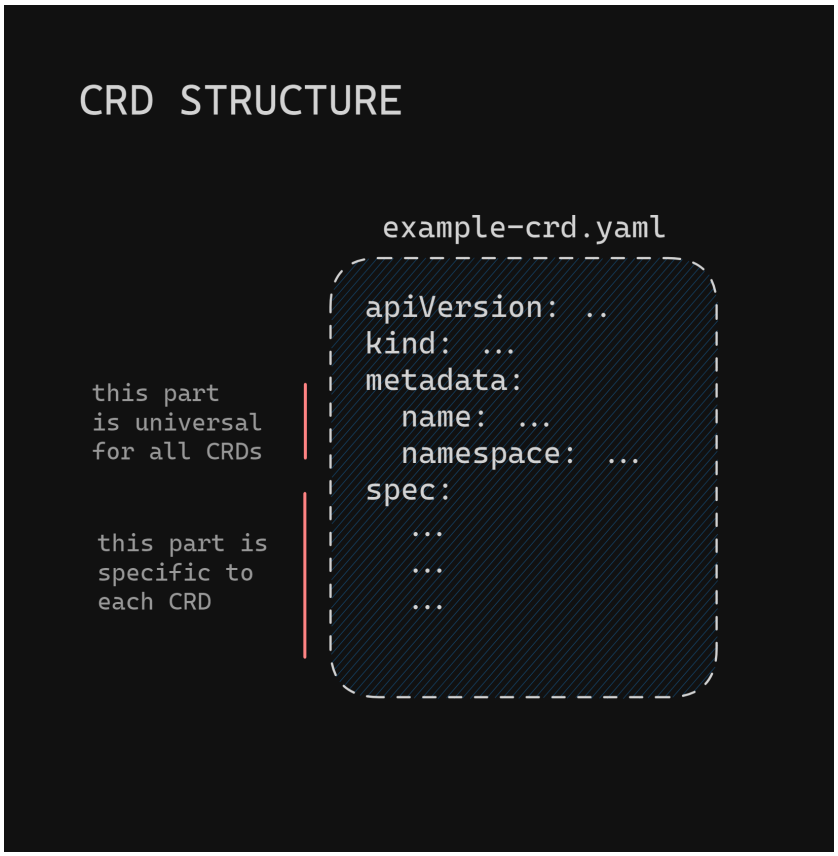

Figure 2.6: CRD example manifest

2.5.3 CNI (Container Network Interface)

- Manages IPs of Pods and Services (for internal communication across all pods)
- Kubernetes creates a NAT network that is 'invisible' and inaccessible to the outside world but enables pods to communicate with each other, independently of the node they are located in.
- The CNI is responsible for giving the IPs to the Pods and Services.

- Kube-proxy exists in each Node to implement routing from Services to Pods.
- Kubernetes uses a subnet for all Services and one for all Pods.
  - **Services CIDR:** (usually) 10.96.0.0/12
  - **Pods CIDR:** (in calico) 192.168.0.0/16
    ** with block size = 26, meaning that each Node gets a subset of these IPs for his pods ex. 192.168.1.1/26 → which translates to 64 available IPs for each Node*

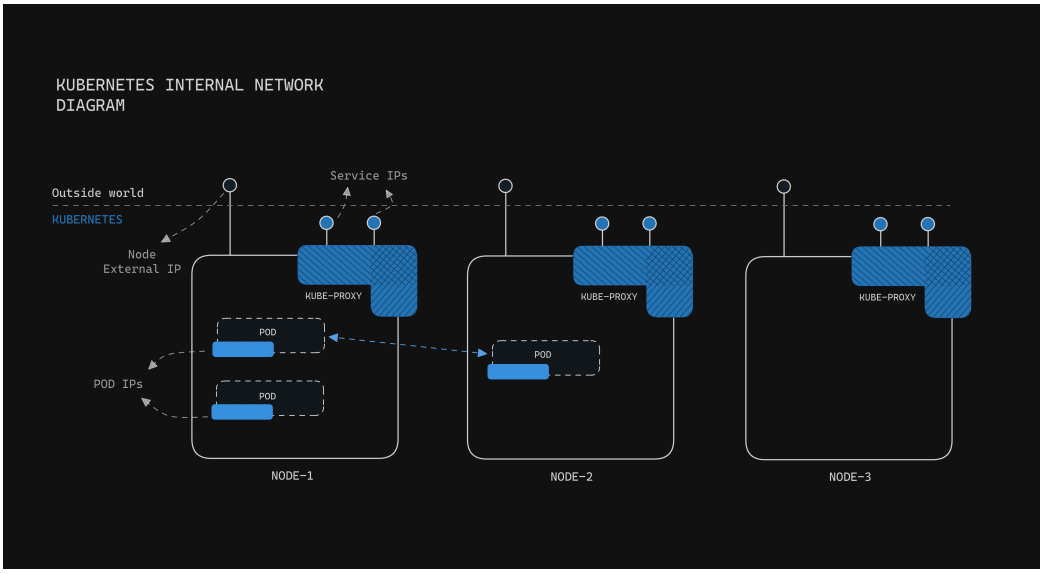

Figure 2.7: Diagram for CNI architecture

2.5.4 Services

A service is an abstraction that is deployed along with Pods, to expose them under a single endpoint. Every Pod has a different and unique IP that changes every time it gets recreated. That creates the need to refer to them, not by their IP address, but in a unified way that represents a container instance or multiple containers clustered together.

Kubernetes Services solves both the problem of dynamic IPs and Load Balancing by offering a single endpoint to connect to Pods in the form of a DNS name which can be accessed from the outside world or from inside the Kubernetes subnet. Services are implemented in the kube-proxy Pods in every Node [11].

The main types of services are 3, with each one of them being an enlarged edition of the previous. Their characteristics are presented below:

ClusterIP

It is the simpler type where Services are accessible only from inside the Kubernetes cluster. Any Pod can communicate with any other Pod from inside the cluster by using its DNS name of the ClusterIP Service. In the diagram below you can better visualize the connectivity between Pods in K8s. Every Service that gets created is at least a ClusterIP meaning that it has to provide an endpoint and also load balancing to the Pods. The default load balancing implementation is usually Round-robin.

NodePort

NodePort allows both internal and external access to a Service. While maintaining the properties of a ClusterIP, it also exposes a port in all Nodes, called the NodePort, thus the name.

The service can now be accessed both from its DNS name and from the IP of any Node plus the NodePort. It does not matter what Node you choose, as all of them have the NodePort exposed and they forward the request to the appropriate Pod, thanks to Kubernetes internal networking.

This Service type is usually used for testing purposes and is not recommended for production use.

LoadBalancer

This Service type augments the properties of NodePort, by allowing external clients to use a single IP as an endpoint. This eliminates the problem of one of the Nodes going down or changing IP and therefore not being able to respond to the client through the NodePort.

When the cluster is deployed at a cloud provider the IP is automatically given. However, when we have a local deployment, we either have to use MetalLB to reserve IPs from the network or enter an IP manually in the Service manifest of the LoadBalancer (for example the IP of one of the Nodes). [12]

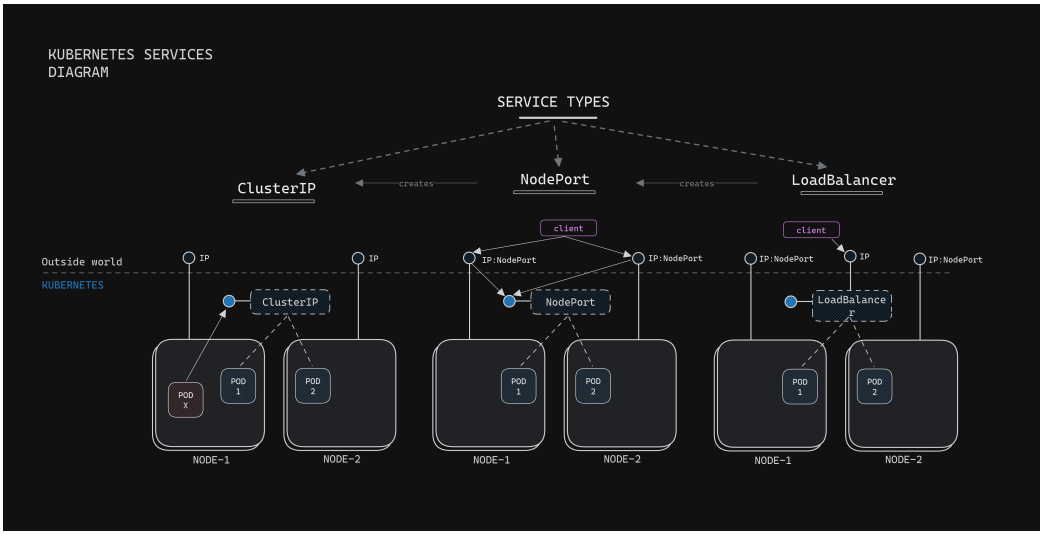

Figure 2.8: Diagram of architecture for types of Services

Ingress

Ingress is a Kubernetes-defined object (CRD) that manages access to the Services from outside the Cluster, typically HTTP connections. It acts like a proxy by providing load balancing, name-based virtual hosting, and SSL termination.

Ingress rules are fulfilled by an Ingress Controller that is running as a Pod and is the one that handles the routing of the requests to the equivalent Service. Ingress Rules are applied to the controller to determine what Service should respond depending on the URL of the request. For example, an Ingress Rule might state that connecting to the /test-route-1 should access the 'Service 1', while another Ingress Rule might state that test-route-2.app.com should access the 'Service 2'.

Ingress Controller gets exposed as a LoadBalancer and this means that we get all the benefits of it such as the round-robin-based Load Balancing that is performed across the Services.

It can also provide secure access to HTTPS endpoints because it can talk with a certificate-generating authority like lets-encrypt which can create and refresh valid certificates. In our work, we use the cert-manager service which runs inside a Pod and manages everything about https certificates. [13]

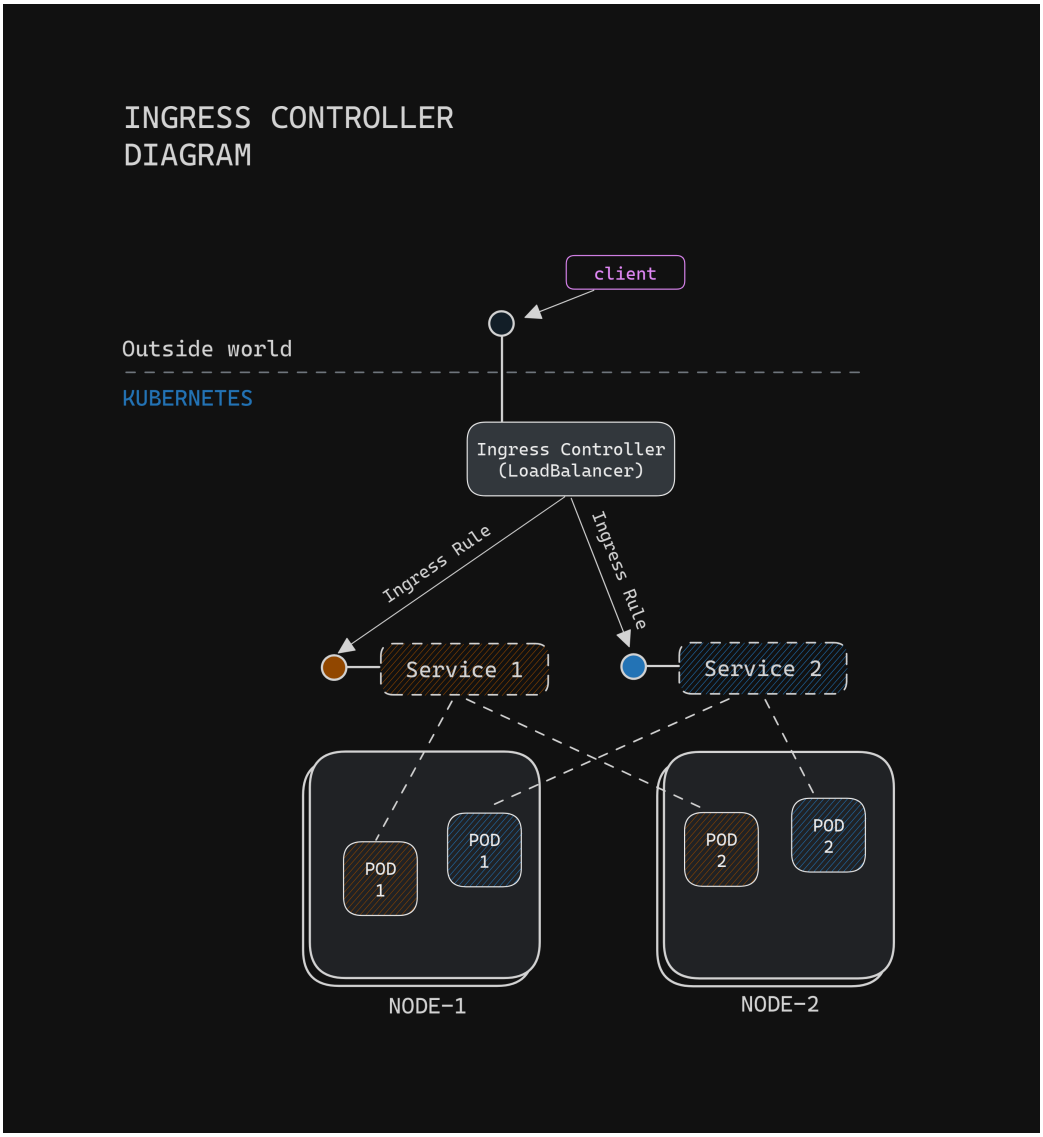

Figure 2.9: Diagram of architecture for Ingress

2.5.5 Workload Resources

A workload is an abstraction we use to describe an application that runs on Kubernetes, whether it runs as a single component or several that work together. In K8s, the smallest deployable unit of computing is called a Pod and it represents a set of running containers. The Pod acts as the 'logical host' of these containers, as it provides an IP and ports. These containers that run inside a Pod are tightly coupled and share the same storage and network

resources. Pods on their own have a predestined lifecycle, because when the Node that they are deployed on goes down, or when they are terminated for any other reason, then they are declared as 'failed' and cannot come back up on their own, thus requiring manual recovery. For that purpose, there are 'logical' entities called Workload Resources that cluster these Pods together and supervise their life cycle. Each of these resources has a controller that is constantly checking for the state and health of each pod and can take the appropriate actions to recover the deployed application in case of a failure. A lot of such predefined controllers are deployed inside the Kube-controller-manager Pod which resides in the control plane, as mentioned earlier [5].

Below, we analyze the most common Workload Resources that are used in Kubernetes and which are also used in the implementation part of this work. The main differentiating factor among the Workload Resources below is the type of application that we want them to encapsulate. For example, some applications are stateless, meaning that they are not dependent on any permanent storage, while others need to claim some storage from the operating system to operate.

ReplicaSet

It is the simplest and most common way to run an application. It is made for deploying and managing stateless application workload on the cluster, where any Pod is identical and interchangeable. A ReplicaSet is deployed using a Deployment object in Kubernetes, in which someone declares the number of copies that the workload should have, and in case any of these Pods go down or the Node goes down, the ReplicaSet controller is responsible for creating a new Pod on one of the available Nodes of the cluster.

StatefulSet

As the name declares a StatefulSet is a resource made for deploying stateful applications. All Pods that are under a StatefulSet have a distinct identity and their main characteristic is that they are linked to persistent storage provided by the cluster. This persistent storage is defined as a PersistentVolume and can be created either manually by the cluster administrator or with a StorageClass given a fixed amount of requested storage capacity [14]. If one of the Pods goes down, the PersistentVolume will remain intact, and then the Pod gets created again by the StatefulSet controller, it will get reattached to the same PersistentVolume without losing any of its data stored there. That is why having a distinct is important and also why scaling such a resource can be tricky and difficult.

DaemonSet

There are scenarios when an application needs to run on every machine, but there is no need to run more than one instance on it. For example, the Kubelet is a workload that requires a Pod to run on every Node of the cluster, to supervise the lifecycle of other Pods and more generally it is an application that helps manage the equivalent Node. Such an application is very similar to a system daemon on a classic UNIX system and that is why we deploy such applications as a DaemonSet. You can run a DaemonSet across every node in the cluster, or across just a subset of them (for example, install the GPU accelerator driver only where a Node has a GPU installed) [15].

Creating your own resources and controllers is also possible through the Kubernetes API [16] to extend its functionality for applications that cannot be deployed with the existing ones [17]. Creating new controllers and CRDs for that purpose is described as creating an operator that can be described as a custom controller for a specific application that follows the Operator Pattern guidelines [18].

2.6 Distributed Systems

Distributed systems are networks of independent components working together to form a cohesive system, designed for improved performance, scalability, and resilience compared to centralized systems. Within this broad spectrum, a fundamental distinction exists between stateless and stateful systems.

A stateless distributed system is one in which the servers do not retain any internal state between requests — each interaction is processed independently, without reliance on information from previous interactions. This absence of state enables such systems to scale horizontally with ease; additional servers can be added to the pool to handle the increased load, and since there's no need for synchronization of state, these new servers can start processing requests immediately. What makes a system truly distributed in this context is its ability to work in concert with other systems, often over a network, to deliver a service. Such systems are robust and resilient, as the failure of a single component often does not impact the availability or performance of the overall system.

Conversely, stateful distributed systems present a more complex scenario. These systems maintain state across requests, remembering previous inter-

actions or the status of operations. This state can include data such as user sessions, information caches, or persistent data required for operations. In such environments, distribution becomes a challenge due to the necessity of data consistency and replication across nodes. Ensuring that all nodes see the same state is non-trivial and often involves sophisticated synchronization and consensus algorithms. The replication of data, which is vital for resilience and fault tolerance, introduces latency and potential bottlenecks that must be managed. Stateful systems require intricate strategies to handle partitioning, replication, and transactional integrity, making the scaling process more complex compared to their stateless counterparts. Despite these challenges, stateful systems are indispensable for scenarios where data continuity and stateful interactions are necessary.

Consensus

Consensus plays a pivotal role in the operation of distributed systems, which are composed of multiple nodes that must coordinate to accomplish a shared objective. This process requires that all nodes in the network reach a unanimous agreement on specific data or the state of the system, despite potential disruptions or malfunctions within the network.

A real-life example of consensus in a distributed system can be seen in a banking application. Suppose a customer deposits into their account. The deposit must be recorded in the bank's database, and all nodes in the system must agree on the account balance. To achieve consensus, the nodes must exchange messages to ensure that all nodes have the same view of the account balance. If one node fails or behaves incorrectly, the other nodes must be able to detect this and continue to operate correctly [19].

Chapter 3

Design & Concepts

3.1 EHS Architecture

We intend to design a system that receives MQTT messages from multiple sensors at the same time and distributes them across multiple microservices. Thus, the first component we have to use is an MQTT Broker. We choose to avoid receiving the incoming messages directly with Apache Kafka, to ease its load and be able to transpose the message data part before it reaches the Kafka Brokers.

Kafka is the central part of our system that processes and persists all data that come from the sensors. Each message starts from the MQTT Broker, then receives a timestamp, enters the Kafka Brokers, and from there, it gets stored in multiple databases with Kafka Connectors. Kafka connectors are deployed in Kafka Connect which is a Kafka-compatible solution for implementing high-throughput data transfers from and to Kafka.

In our case, we have 3 main database types where we want Kafka to dump our data. We have a time series database where we want to store data for only a short period to ease the load from the central database for real-time time series data.(The data fetching from the time series database is being performed by a Hasura Deployment which uses GraphQL to perform the data queries and automates the API creation process [20]). Then we have a no-sql database with the purpose of storing aggregated data from the sensors. The continuous data streams get aggregated, to avoid storing more information than needed, and get written in the no-SQL database permanently. Lastly, we need a combination of the above two databases to store every sensor data and metadata possible to use for the training of a machine learning model. Thus, we have to deploy a data lake with much more storage than the rest.

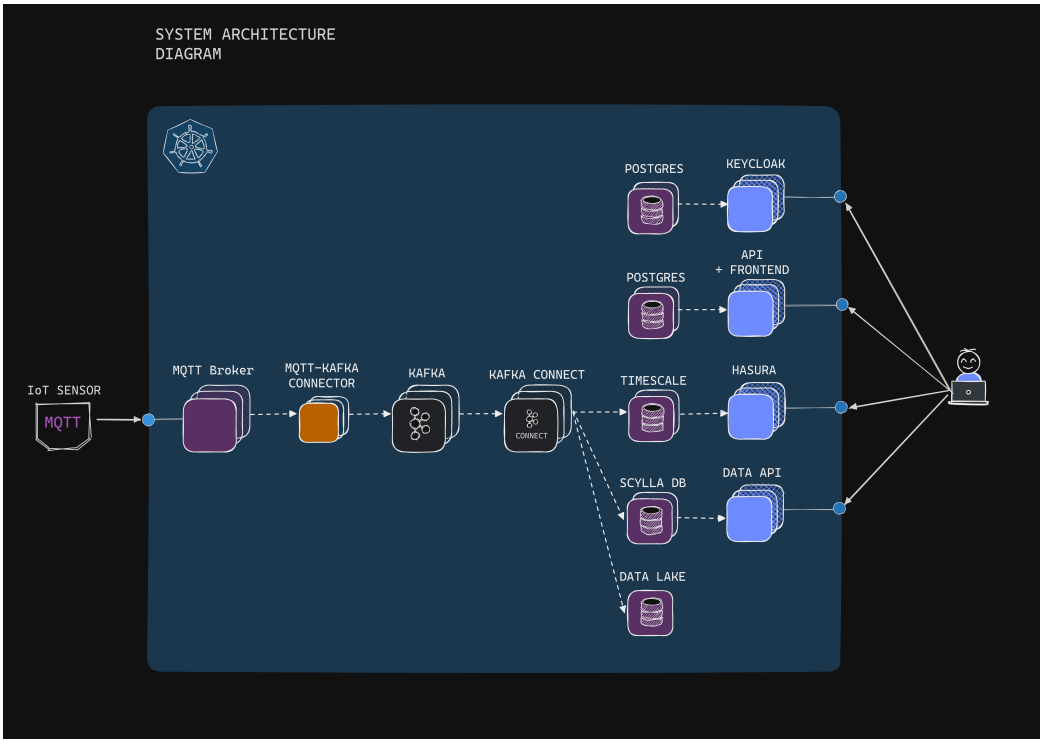

Figure 3.1: Diagram for architecture of the IoT system

Below, we present every component of our system and how that operates in a distributed manner.

3.2 Why choose Kubernetes

The capacity to manage and serve innumerable device connections, convey massive amounts of data and deliver high-end services such as real-time analytics involves a deployment architecture that can dynamically scale up and down in response to IoT deployment demands. Kubernetes enables developers to autonomously scale up and down across various network clusters. Many IoT solutions are classified as business/mission-critical systems that must be extremely dependable and available. As an example, an IoT solution crucial to a hospital's emergency healthcare facility must be available at all times [21]. Kubernetes provides developers with the tools they need to deploy highly available systems.
Kubernetes' design also allows workloads to run independently of one another.

On a practical level, Kubernetes is the most well-known and most widely used system for container orchestration. Possibly, it is the only open-source and production-ready with a relatively long history of updates and bug fixes. Often, there is a comparison between Kubernetes and Terraform. However, those two systems solve different problems. While Kubernetes focuses on running and orchestrating multiple containers, Terraform targets the Infrastructure as Code (IaC) space.

3.3 MQTT Broker

IoT networks deploy several radio technologies at the lower level. Lower-level communication protocols may include LoRaWAN, SigFox in long-range (km)-low data rate (bps-kbps), cellular/4G/5G in long-range(km)-high data rate (Mbps), Zigbee, Zwave in medium range (m)-medium-data rate (kbps) and WiFi in medium range(m)-high-data rate (Mbps). Apart from the radio technology used to transmit the data, the appropriate messaging protocol also plays a key role. MQTT, CoAP, AMQP, and HTTP are the four widely accepted and emerging messaging protocols for IoT systems. In this work, we work with the MQTT message protocol as it is designed specifically for minimizing network bandwidth and power consumption, while also ensuring reliable delivery over low-bandwidth and unreliable networks [22].

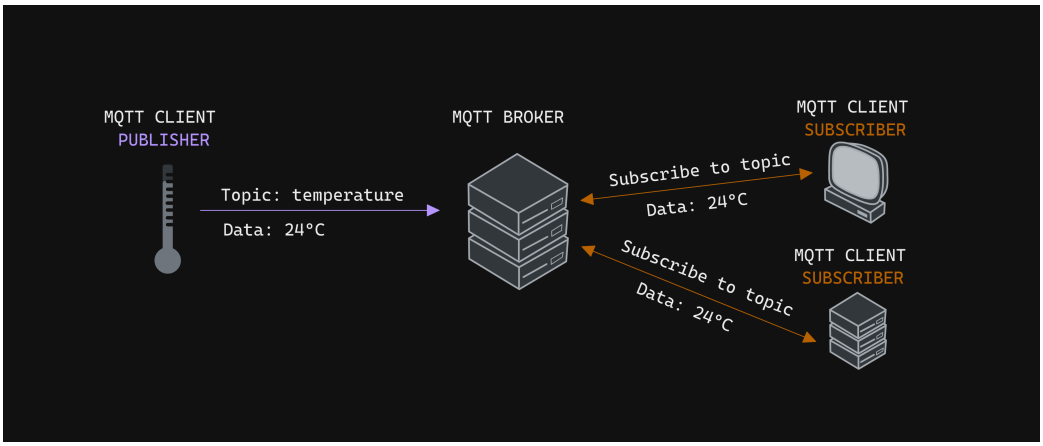

Figure 3.2: Diagram for Pub/Sub in MQTT

Apart from being efficient, it is also a relatively flexible protocol as it offers three qualities of service (QoS) for message delivery which allow the system developer to make the tradeoff between a high message delivery suc-

cess rate and high throughput. Each QoS level is usually targeted for a specific category of applications.

For example, QoS 0 (or known as 'at most once') is used for sending real-time sensor data, like humidity, pressure, or power consumption, where throughput is important and the loss of messages is acceptable. Continuing we have QoS 1 (known as 'at least once') where the message is going to be delivered to the broker at least one time, which allows message duplicity in case the broker message acknowledgment gets lost but ensures the message gets delivered. An example used is a command and control system like a home automation system, where we need the device to receive and execute the command. We can understand that because these actions are human-operated, the message frequency is not as great as the sensor measurements. Finally, we have the QoS 2 (also known as 'exactly once') which is useful in billing systems or remote surgery applications, where duplicate messages are not acceptable.

In our case, we can easily conclude the use of QoS 0 for our sensory data streams, as we see that it reduces both memory usage, for temporary storing of messages, and latency, as there is no need for the publisher to wait for a response from the broker.

For future work, we will consider using QoS 1 to communicate with home devices such as an AC or a power plug to operate them remotely. We should note that MQTT is a bi-directional communication protocol. This helps in both sharing data, managing, and controlling devices.

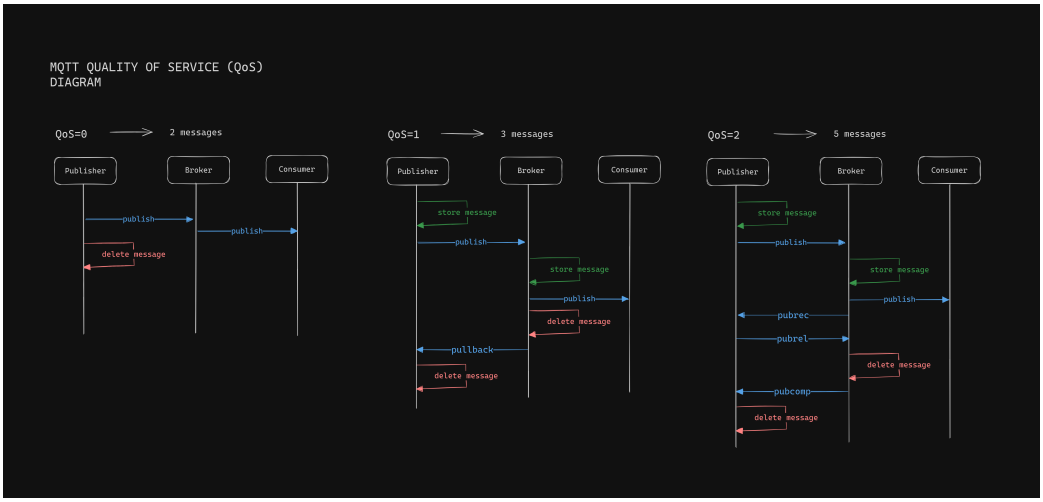

Figure 3.3: Process diagram for QoS in MQTT

The components of an MQTT system are the following:

- Publisher or Producer (client)
- A broker (server)
- Consumer/ Subscriber (client)

An MQTT server is a program or device based on MQTT that acts as a post office between publishers and subscribers, while an MQTT client is a program that either sends or receives messages from the MQTT server. Each message contains payload data, a QoS, a collection of properties, and a topic name.

3.3.1 Choosing an appropriate MQTT Broker

We initially tried the Eclipse Mosquitto Broker [23] which is a lightweight, commonly used, and well-maintained broker software. However, the fact that it is a single-threaded and not a horizontally scalable implementation made it difficult to incorporate it in an everything-distributed data processing pipeline. Therefore, we set the requirements for a highly scalable broker that implements load balancing across all nodes and is fault tolerant.

Looking across the open-source and commercially licensed options[22] we considered the EMQ X [24], the HiveMQ **hivemqMQTTBroker**, and the VerneMQ [25] brokers.

Features	Source code availability model		Design and Implementation			
MQTT Brokers	Open/Close	Software License	Written in	Supported OS	Latest stable release, release date	Developed by
Mosquitto	Open	Eclipse Public License 1.0, Eclipse Distribution License 1.0 (BSD)	C	Linux, Mac, Windows	1.6.9, 2020-02-27	Eclipse Foundation
Bevywise MQTT Route	Close	Commercial License	C, Python	Linux, Unix, MacOS, Windows, Raspbian	2.0, 2019-12-03	Bevywise Networks
EMQ X	Open	Apache License version 2.0	Erlang	Linux, Mac, Windows, BSD	3.0, 2019-04-03	EMQ Enterprise Inc.
HiveMQ CE	Open	Apache License version 2.0	Java	Linux, Mac, Windows	2020.3, 2020-07-06	HiveMQ GmbH
HiveMQ	Close	Apache License version 2.0	Java	Linux, Mac, Windows	4.3.5, 2020-07-17	HiveMQ GmbH
IBM WIoTP Message Gateway	Close	Commercial License	C	Linux	5.0.0.1, 2019-02-29	IBM
JoramMQ	Close	Commercial License, LGPL	Java	Linux, Unix, MacOS, Windows, Raspbian	1.13, 2019-04-29	ScalAgent
flespi	Close	Commercial License	C	-	-, 2018-04-05	Gurtam
PubSub+	Close	Commercial License, Free Version	C, C++	Linux, Mac, Windows	8.13, 2018-09-28	Solace
Thingstream	Close	Commercial License	C, C++, Java, JavaScript, Python, Go	-	3.3.0, 2019-03-14	u-box
VerneMQ	Open	Apache License version 2.0	Erlang	Linux, Mac	1.9.1, 2019-08-12	Octavo Labs AG
RabbitMQ	Open	MPL 1.1	Erlang	Linux, Unix, Mac, Windows, BSD	3.8.1, 2019-10-31	Pivotal Software
ActiveMQ	Open	Apache License version 2.0	Java	Windows, Unix, Linux, Cygwin	5.15.10, 2018-09-01	Apache Software Foundation
ActiveMQ Artemis	Open	Apache License version 2.0	Java	Windows, Unix, Linux, Cygwin	2.10.1, 2019-09-26	Apache Software Foundation

Figure 3.4: Available MQTT Broker software comparison table

We decided to use the HiveMQ, because of better documentation, active and high maintenance, and easier integration with Kubernetes through the available Operator. Operators are control loops that can handle stateful applications in Kubernetes. They can handle dynamic recovery of replicas as well as upscaling and downscaling of the cluster. In our case especially they should be able to do that without sacrificing broker availability.

HiveMQ stores the data in memory which is a choice that allows faster read & write operations, which is important in MQTT message handling. We want the MQTT broker to act solely as an intermediate for sensor data.

For further persistence of all messages, we connect the HiveMQ cluster with a Kafka Broker cluster, which can handle longer message queues and hold the data for longer periods.

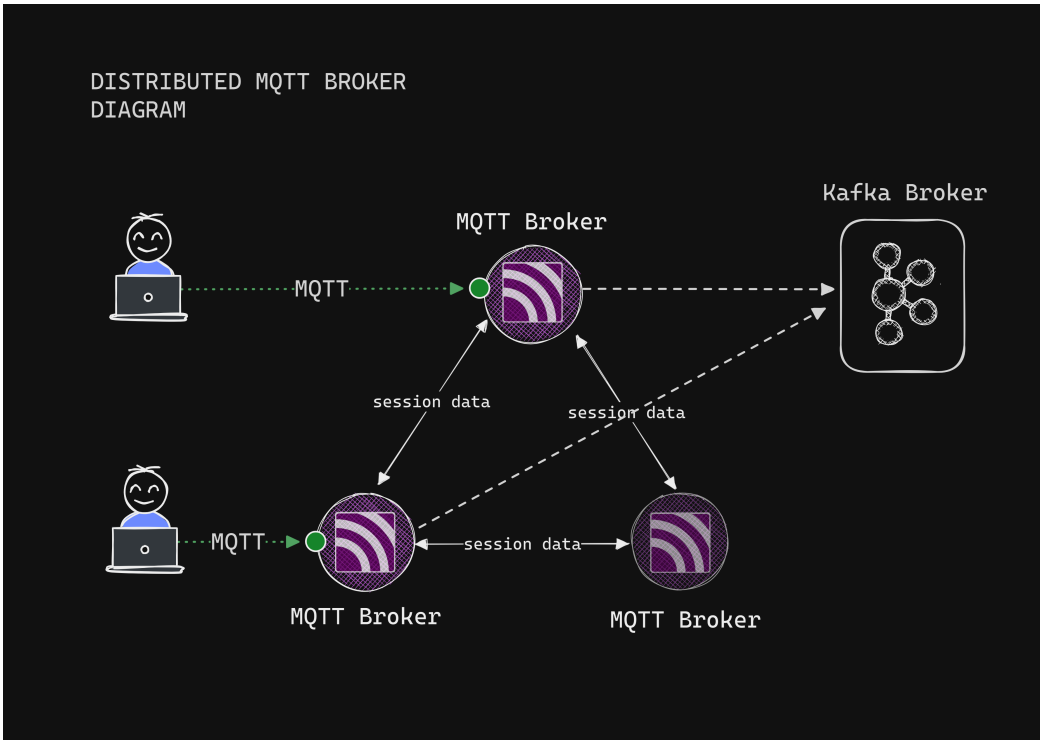

Figure 3.5: Diagram of architecture for distributed MQTT Brokers cluster

3.4 Kafka Broker

Apache Kafka [26], an open-source software, is a distributed and fault-tolerant event streaming platform based on the publish/subscribe model. Following topic-based design and written in Scala and Java it is currently maintained by Apache foundation and is used in a majority of Big Data solutions and real-time data pipelines.

The publisher/subscriber model implies that there are 2 distinguished types of clients. The ones that produce/publish messages to the Kafka cluster of brokers, which are called Producers, and the ones that consume those data based on a classification that uses topics to describe, which are called Consumers.

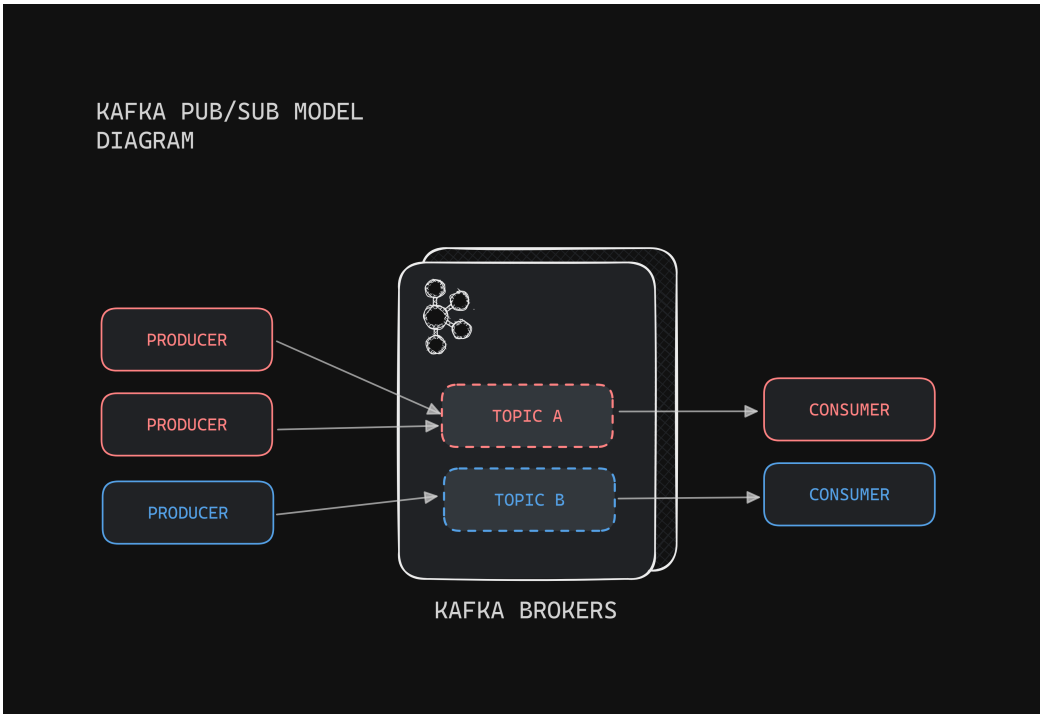

Figure 3.6: Diagram for Pub/Sub in Kafka

Kafka Brokers are responsible for maintaining and distributing the messages to clients.

To store and manage all the configuration data of the broker's cluster, meaning the information about the cluster nodes, the message topics, and partitions, Kafka relies on a third-party service. The most popular, because of its adaptation is Zookeeper [27].

However, because Zookeeper requires an extra independent deployment object to run alongside Kafka, we will use KRaft, which is a more recently developed alternative, runs on the same Pod as Kafka, and is production-ready.

3.4.1 Distributed Kafka Cluster architecture

Running a distributed system like Kafka means that it has to be highly available and fault-tolerant. That is why every topic is replicated across multiple brokers. One of those active brokers will be responsible for responding to the requests for the specific topic and will be declared as a Leader. The rest will be declared as replicas. As one can assume, in case the Leader goes down, one of the replicas gets elected as the new Leader and Kafka continues to serve requests with minimal downtime.

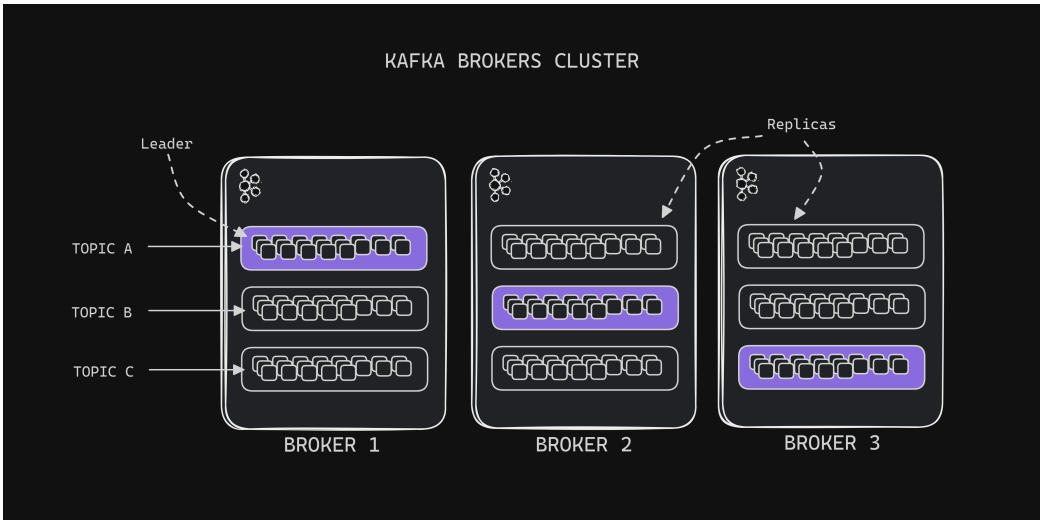

Figure 3.7: Diagram for internal distributed function of Kafka

Topics can also be partitioned across multiple nodes so that they can grow beyond the limits of a certain node and for load balancing of the writing process. Each partition has also a preferred Leader, which is responsible for handling all read and write operations for that partition. All the data concerning the preferred Leader for each topic and partition are stored in Zookeeper/Kraft.

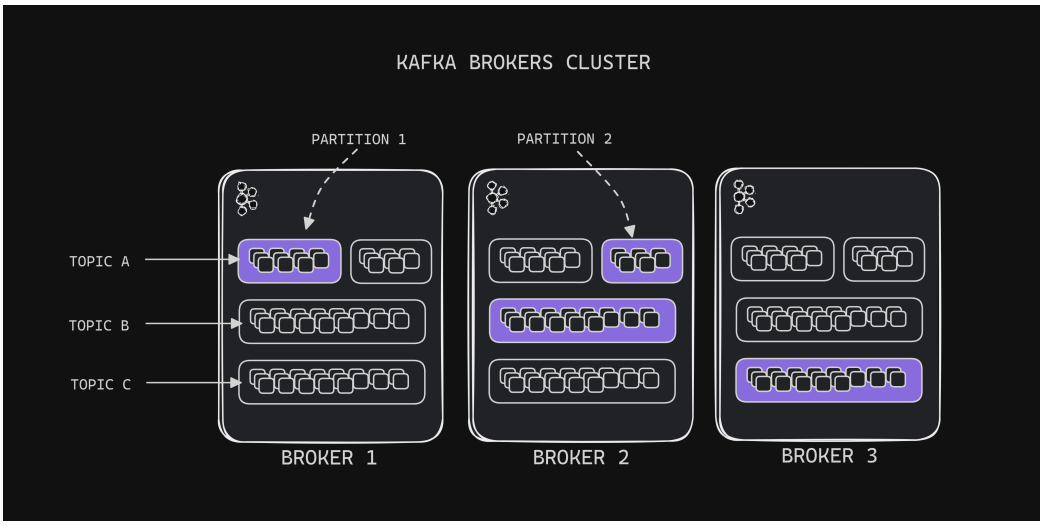

Figure 3.8: Diagram for partitions architecture in Kafka

3.5 MQTT to Kafka Connector

We choose to create a custom Connector to transfer messages from the MQTT Brokers Cluster to the Kafka Brokers Cluster. This choice is mainly to have more control over the messages that derive from the devices. Different devices may send messages with different structures depending on the firmware. Thus, we need more delicate code to process them and convert them to the appropriate format. Plus, we want to experiment with a more simple way of horizontal autoscaling, by using a simple autoscaler, in contrast with Kafka Connect.

The design of our Connector is fairly simple. We deploy an MQTT Client that subscribes to the appropriate MQTT Topic and acts as a Consumer of the messages. After that, follows the message processing.

MQTT messages do not provide a timestamp of arrival and that is why we need to insert a timestamp to our message payload before we produce the message to the Kafka Brokers.

At the same time, we open a connection to the Kafka Brokers and Subscribe to the appropriate Topic as a Producer. Once the messages are converted they are published to this topic.

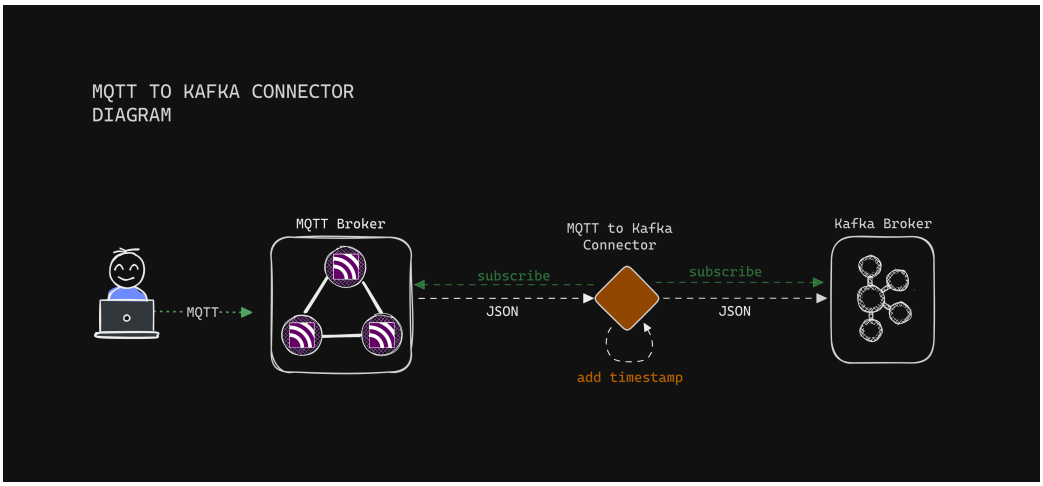

Figure 3.9: Diagram of MQTT-Kafka Connector architecture

By default, the MQTT Broker sends the message to every Client who is subscribed to the following Topic. This means that if we scale the Connector instances horizontally, each instance will transfer the same message multiple times.

To solve this issue, we deploy each MQTT Client with a unique ID, so that all can keep the connection open at the same time and also make use of

the shared subscriptions [28].

Another key characteristic of MQTT Brokers is the 'Shared Subscriptions'. We choose to use MQTT v5 (compared to MQTT v3) in our system because it standardizes the shared subscriptions and allows multiple MQTT client instances to share the same subscription on the Brokers.

Topics can be shared among many clients that belong to the same group. This means, that the MQTT Broker will send each message only once to the Clients of the same group in a round-robin manner, enabling a load-balancing strategy.

What we have to do is make all Clients subscribe to a topic with the following syntax `$share/<group-id>/<topic-name>` .

`$share` : A prefixed name telling the MQTT server this is a shared subscription.

`<group-id>` : A string, without any wildcards ('/', '+', and '#'), that identifies a subscription group. All clients with the same GroupID and Topic are part of the same shared subscription.

`<topic-name>` : A string – can include wildcards ('+' and '#') – that denotes the topic filter to be used. This is equivalent to the topic filter in a non-shared subscription.

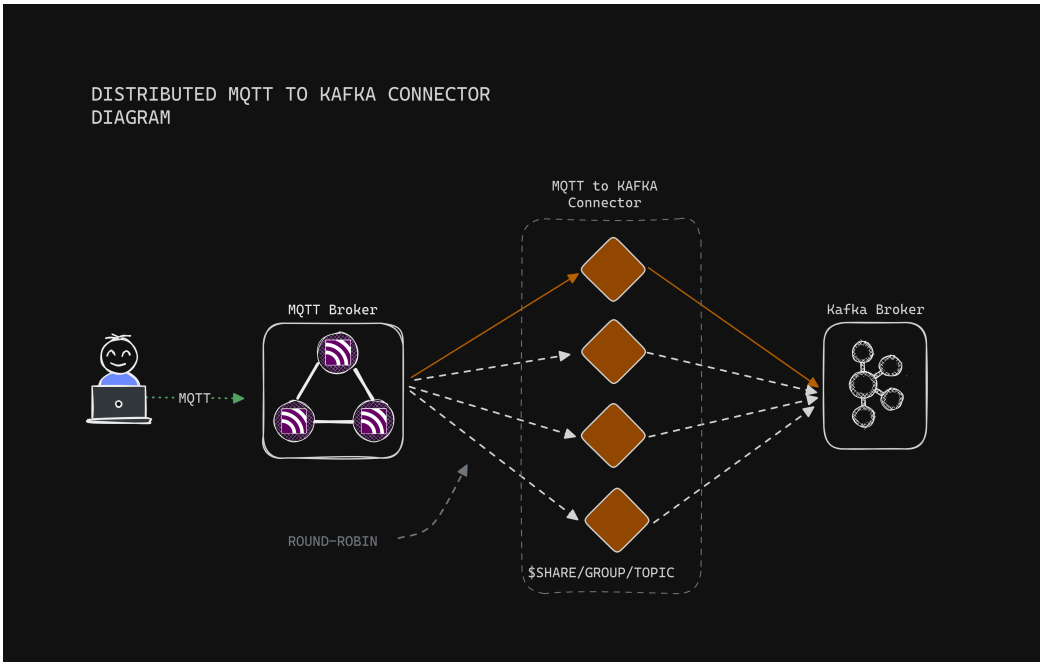

Figure 3.10: Diagram of MQTT-Kafka Connector distributed architecture

3.6 Kafka Connect

Kafka Connect is a tool for scalably and reliably streaming data between Apache Kafka and other data systems. It makes it simple to quickly define connectors that move large data sets in and out of Kafka [29]. Kafka Connect is a server process that runs independently of the Kafka Brokers themselves on different nodes, forming a Kafka Connect Cluster.

3.6.1 Task Rebalancing

When a connector is first submitted to the cluster, the workers rebalance the full set of connectors in the cluster and their tasks so that each worker has approximately the same amount of work. This rebalancing procedure is also used when connectors increase or decrease the number of tasks they require, or when a connector's configuration is changed. When a worker fails, tasks are rebalanced across the active workers. When a task fails, no rebalance is triggered, as a task failure is considered an exceptional case.

3.6.2 Workers

Kafka Connect calls Workers individual processes that are responsible for executing Connectors and Tasks. These Workers can either work run in a distributed manner or not.

- **Standalone Workers**
  A single process is responsible for executing all connectors and tasks. Since it is a single process, it requires minimal configuration. Standalone mode is convenient for getting started, during development, and in certain situations where only one process makes sense, such as collecting logs from a host

- **Distributed Workers**
  In distributed mode, you start many worker processes and they coordinate to schedule the execution of connectors and tasks across all available workers. If you add a worker, shut down a worker, or a worker fails unexpectedly, the rest of the workers acknowledge this and coordinate to redistribute connectors and tasks across the updated set of available workers. Behind the scenes, connect workers use consumer groups to coordinate and rebalance.

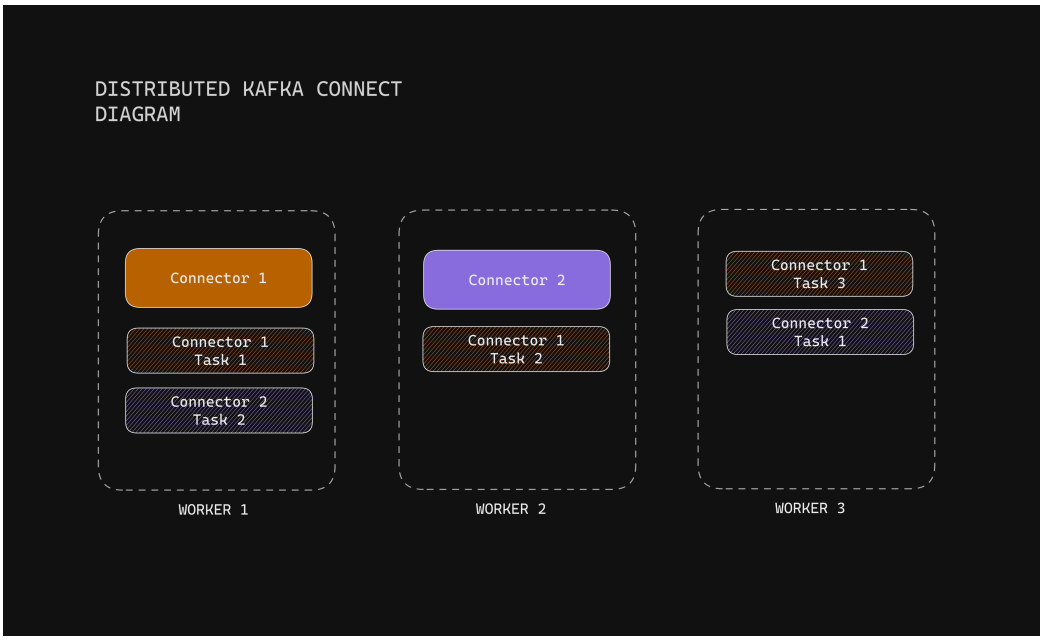

Figure 3.11: Kafka Connect distributed architecture

3.6.3 Connectors

Kafka Connect implements two types of connectors, depending on the direction of the data stream.

- **Source Connector**
  Source connectors ingest entire databases and stream table updates to Kafka topics. Source connectors can also collect metrics from all your application servers and store the data in Kafka topics–making the data available for stream processing with low latency.

- **Sink Connector**
  Sink connectors deliver data from Kafka topics to secondary indexes, such as Elasticsearch, or batch systems such as Hadoop for offline analysis.

3.6.4 Converters

Kafka Connect does not have the ability to understand the message schema, meaning that it wants to have the type and name of each message field declared, in order to transfer it properly. The matching process for the payload fields is done by tasks. Tasks use converters to change the format of data from bytes to a Connect internal data format and vice versa.

Most common converters are the AvroConverter, JsonSchemaConverter and JsonConverter with the latter one requiring no Schema registry [30].

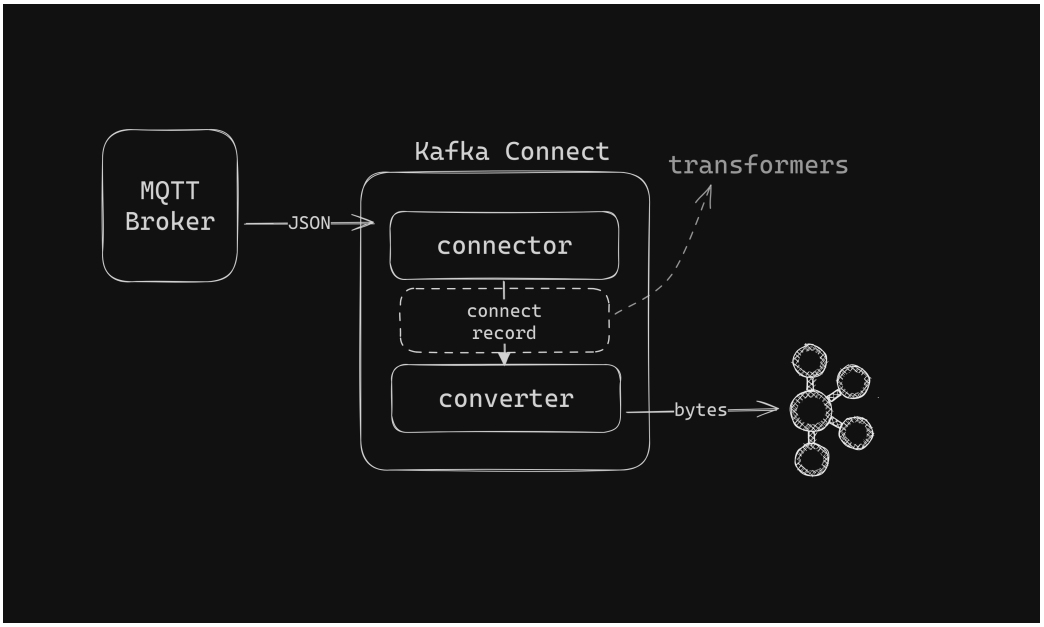

Figure 3.12: Schema of Kafka Connector internal components

3.6.5 Schema Registry

A schema defines the structure of message data. It defines allowed data types, their format, and their relationships. A schema acts as a blueprint for data, describing the structure of data records, the data types of individual fields, the relationships between fields, and any constraints or rules that apply to the data.

Schema Registry enables you to define schemas for your data formats and versions and register them through the registry. Once registered, the schema can be shared and reused across different systems and applications. When a producer sends data to a message broker, the schema for the data is included in the message header, and Schema Registry ensures that the schema is valid and compatible with the expected schema for the topic [31].

It is a standalone server process that runs on a machine external to the Kafka brokers. Its job is to maintain a database of all of the schemas that have been written into topics in the cluster for which it is responsible. That "database" is persisted in an internal Kafka topic and cached in Schema Registry for low-latency access. Schema Registry can be run in a redundant, high-availability configuration, so it remains up if one instance fails [32].

3.7 Kafka to TimescaleDB Connector

For distributing the data from Kafka to the Timescale database we use a Connector deployed to Kafka Connect. To use a Schema Registry the data need to be transformed into serialized format (ex. Avro). Because Kafka does not provide data conversion into such formats, it means that for Kafka Connect to receive a serialized message, that message needs to be converted in some previous step.

The 2 solutions are presented in the diagram below. The goal is to output the message from Kafka into an Avro format. Thus, we either can convert the message inside the MQTT-Kafka Connector or we can use another Kafka Connector which will consume the message from Kafka and produce another message in a different topic.

Because each Kafka Connector needs a different message format depending on the database type, the second design is preferable because we can transpose the messages accordingly for each database.
(For our testing we avoid using a Schema Registry as we can guarantee the integrity of the structure of the message, so we use a simple JSONConverter [33].)

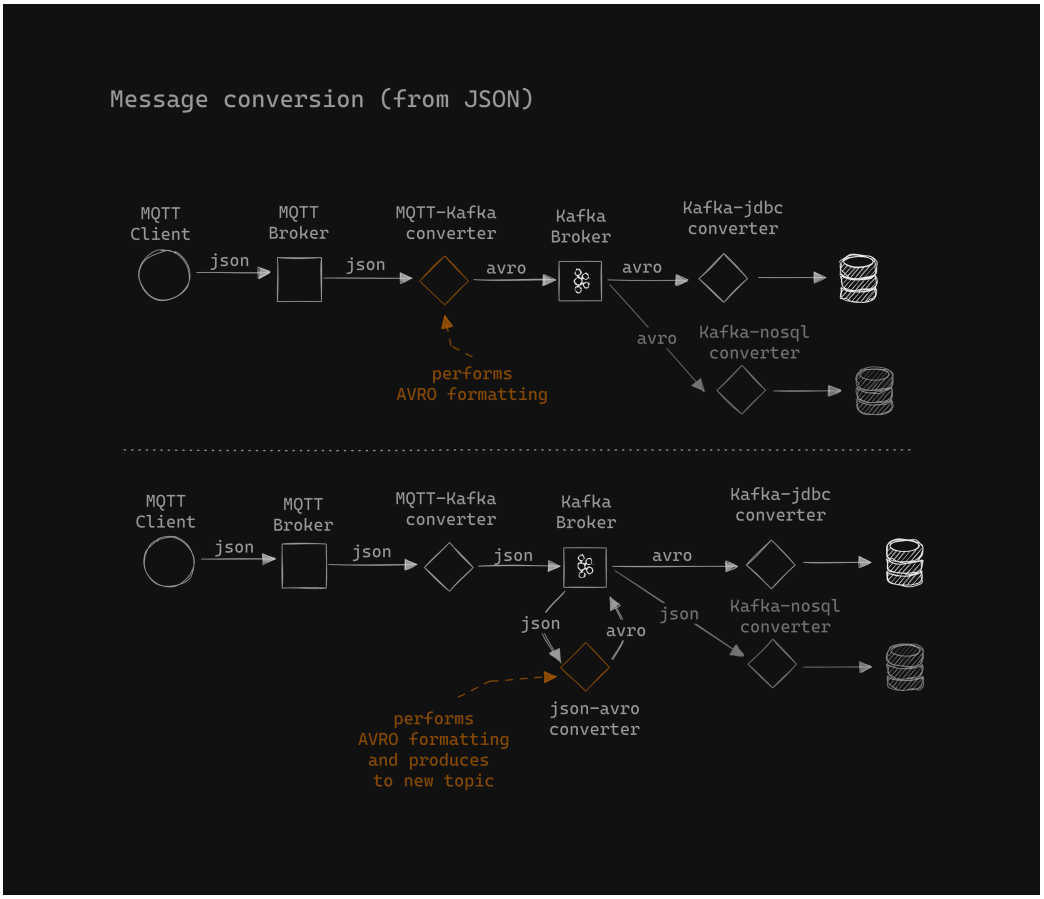

Figure 3.13: Diagram of message conversion solutions

3.8 TimescaleDB / Postgresql

High availability is a top priority for every component of our system. This is why we chose Timescale for our time series database, which is an extension of the PostgreSQL database and can run in a distributed manner.

The main function of this database is to hold real-time data from sensors, mainly to feed the graphical interface of the application. This allows us to ease the load from our primary database and also avoid storing permanently very detailed information about the sensors.

3.8.1 Replication

Timescale offers High Availability (HA), by providing multiple places to get the same data and by eliminating single points-of-failure. The way it does that is by replicating the data on the primary database to one or more read-only replicas. The replication is performed with streaming replication [34] which uses WAL files to sync files from the primary database across the replicas. While a client can perform a write to only the Primary database, the reads are distributed across all Replicas in a load-balancing manner [35].

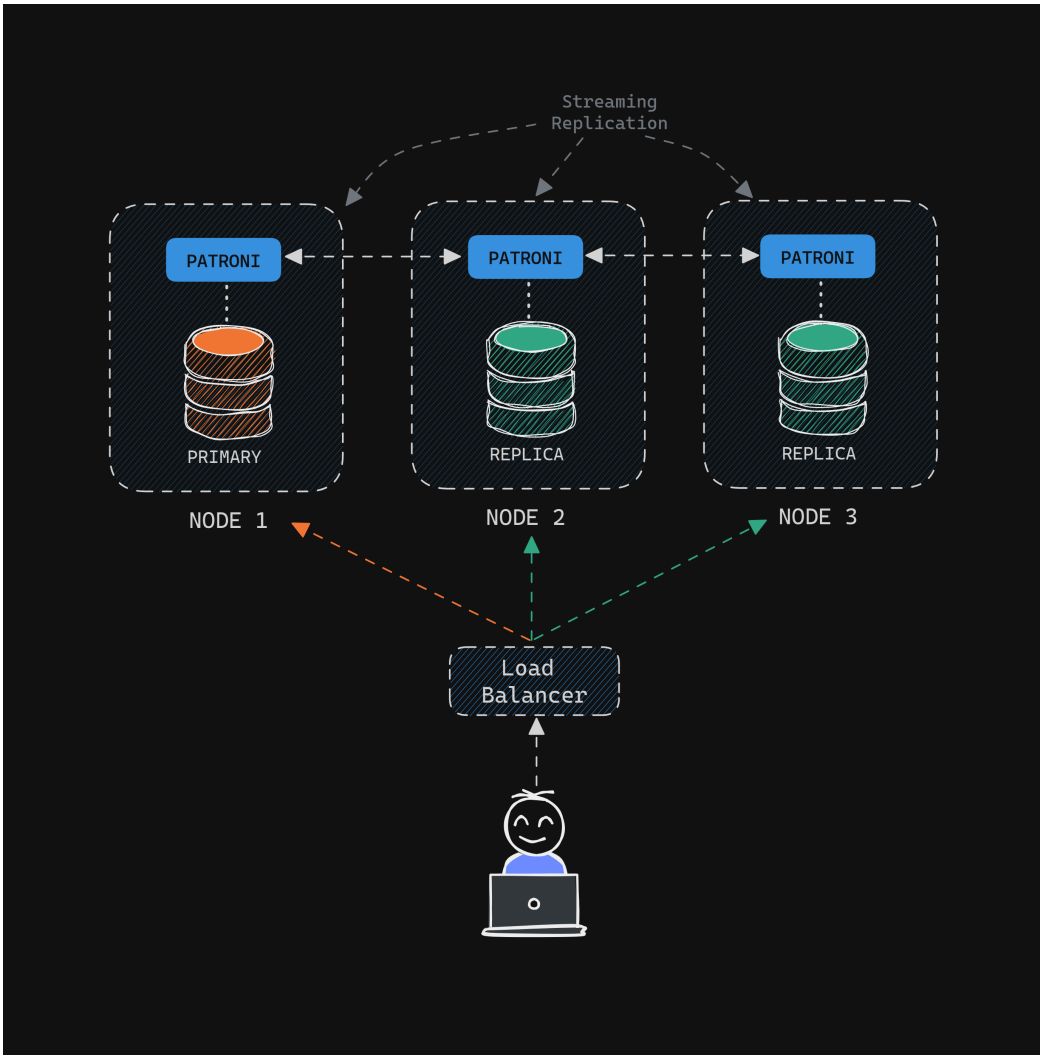

Figure 3.14: Architecture of distributed postgresql deployment

3.8.2 Failover

Timescale relies on Patroni which acts as a failover solution, in order to run in distributed mode. When the Leader goes down, we need to keep responding to write requests, and for that, a new Leader needs to be elected. The leader election process is handled by Patroni, which uses an etcd cluster to store all the configuration data. Etcd implements the leader election with the Raft consensus algorithm [36]. When deployed in Kubernetes Patroni is able to use Kubernetes objects in order to access the existing etcd cluster and store the state of the cluster there, thus eliminating the need to operate an extra Etcd deployment [37].

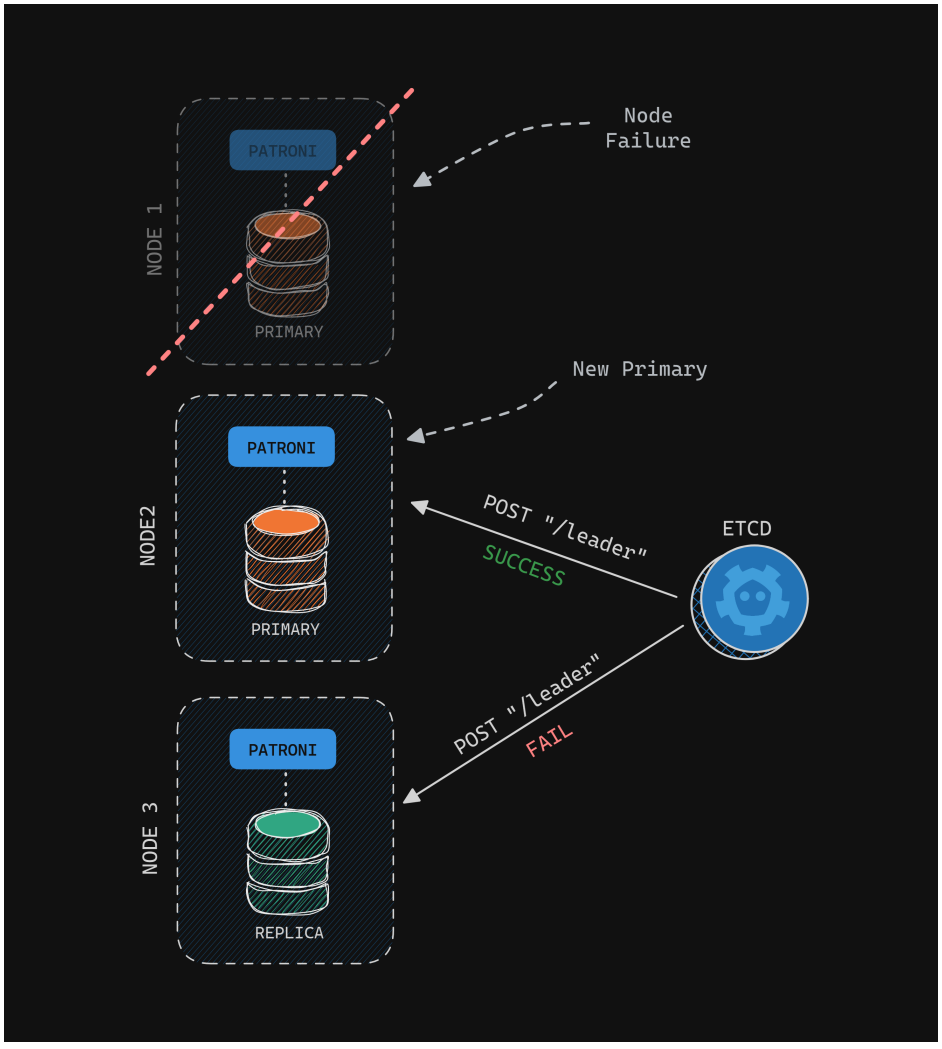

Figure 3.15: Leader election diagram for postgresql (Raft)

3.8.3 GraphQL

To access our time series data and make them available to the frontend application we make use of GraphQL. GraphQL is a query language for APIs and a runtime for fulfilling those queries (GraphQL [38] was originally developed by Facebook and released publicly in 2015). To deploy it, we make use of the Hasura GraphQL engine, which gives us high-performance GraphQL for any Postgres database without having to write any backend code.

By using TimescaleDB and Hasura GraphQL engine together, we are able to query for real-time data using the Subscriptions operation. It enables you to subscribe to events on the server and get real-time updates for each record insertion, modification, or deletion [39]. Usually, this is being done using WebSockets, but GraphQL Subscriptions eliminate that need.

3.9 Deployments

To deploy the simpler stateless applications, such as the Frontend and Backend of our system, in a distributed way, we employ the Deployment object in Kubernetes which creates a ReplicaSet. Deployments are a declarative way to manage the instances of the applications, ensuring a specific number of pod replicas are running at all times. The number of replicas can vary depending on the load of the application and if we want the application to scale automatically in real time we can use an autoscaler like KEDA (Kubernetes-based Event-Driven Autoscaler), which is able to scale up or down the system as needed based on metrics such as CPU utilization or network bandwidth.

To function truly as a distributed application, each Deployment should be accompanied by a LoadBalancer Service, which will distribute the load across all instances with a round-robin type of load-balancing algorithm.

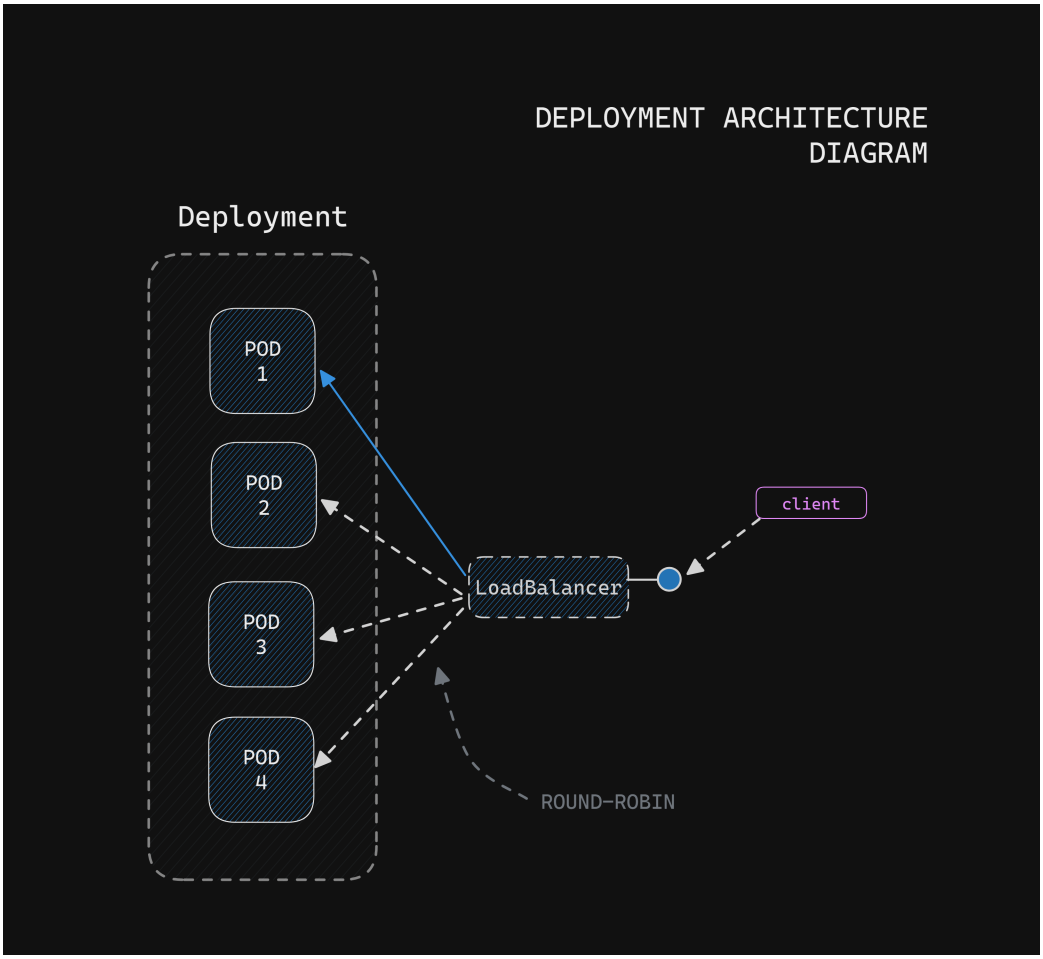

Figure 3.16: Deployment diagram deployed in Kubernetes

An example of a Deployment object that uses a specific Docker image and exposes port 80 can be seen below:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
        image: your-frontend-image
        ports:
        - containerPort: 80
```

3.10 How we expose the services

In Kubernetes, iptables is replaced by eBPF [40], as demonstrated in the creative. The most basic type of load balancing in Kubernetes is load distribution, which is simple to implement at the dispatch level, meaning that simple algorithms such as round-robin are used.

Kubernetes has two load distribution mechanisms, both of which operate through a feature called Kube-proxy, which maintains the virtual IPs used by services. Kubernetes provides a basic, yet effective, form of load balancing by distributing incoming traffic among the available instances of a service, and this distribution is implemented at the dispatch level, ensuring a more equitable sharing of the load.

In Kubernetes, each Pod receives its own unique IP address, reachable from any other pod in the cluster, whether colocated on the same physical machine or not. This requires advanced routing features based on network virtualization [41].

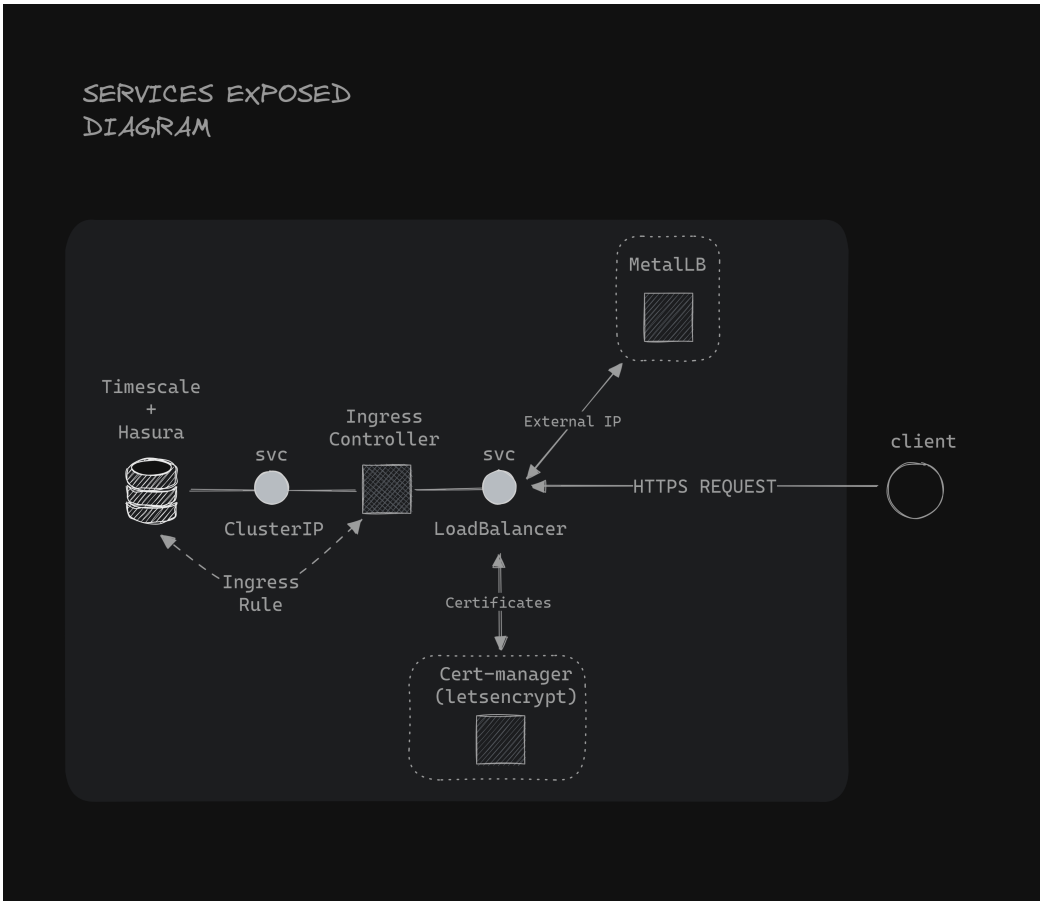

Figure 3.17: Diagram of the eposure of Services to the oustide world

3.11 HELM Charts

A Helm Chart is a collection of files organized in a specific directory structure. At its core, a Helm Chart contains a `Chart.yaml` file that provides metadata about the chart, a set of default values in a `values.yaml` file, and templates that generate Kubernetes manifest files. These templates, written in the Go template language, reference the values defined in `values.yaml` or values passed at runtime, allowing for customizable deployments. Additionally, Charts can include a `charts/` directory containing dependencies, which are Charts that your Chart relies on, and a `templates/` directory for template files [42][43].

The true power of Helm Charts lies in their ability to package the entire lifecycle of Kubernetes applications. They encapsulate all necessary Kubernetes resources and configuration into a single cohesive unit that can be easily

distributed, versioned, managed, and updated.

For example, if you're deploying a web application on Kubernetes, you could use a Helm Chart to define not just the Deployment and Service resources needed to run your application, but also more complex aspects like Ingress rules, Persistent Volumes, and ConfigMaps. With Helm, you can then install this Chart into your Kubernetes cluster with a single command, `helm install my-web-app ./my-chart`, where `my-web-app` is the release name, and `./my-chart` is the chart directory. This not only simplifies the initial deployment but also streamlines updates with `helm upgrade`, rollbacks with `helm rollback`, and customizations through different environments by overriding default values.

```
wordpress/
  Chart.yaml          # A YAML file containing information about the chart
  LICENSE             # OPTIONAL: A plain text file containing the license for the chart
  README.md           # OPTIONAL: A human-readable README file
  values.yaml         # The default configuration values for this chart
  values.schema.json  # OPTIONAL: A JSON Schema for imposing a structure on the values.yaml file
  charts/             # A directory containing any charts upon which this chart depends.
  crds/               # Custom Resource Definitions
  templates/          # A directory of templates that, when combined with values,
                      # will generate valid Kubernetes manifest files.
  templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes
```

Figure 3.18: Example structure of Helm Chart files

The main reason we use Helm Charts in this deployment is because they offer the easiest way to install and uninstall many CRDs under a common tag, which is very convenient for testing different alternatives for the same component. Helm charts also offer an easier way to maintain and manage the various deployments, because all are categorized based on their tag.

Chapter 4

Implementation

4.1 Infrastructure

In this work, we consider a homogeneous 3-node Kubernetes cluster composed of VMs running on-premises. Each VM is equipped with 2 CPUs, 10Gb of RAM, and 1TB of persistent storage. Each VM runs on top of Hyper-V hypervisor with Ubuntu Server 22.0 installed as the Operating System. To make our VMs ready to act as nodes in the Kubernetes cluster we need to apply some prerequisites. Firstly, we assign each machine a public IP and a hostname. Then, for each machine, we add the IPs and hostnames of the rest to their DNS records.

Each VM is called a Node in the Kubernetes cluster and these Nodes are partitioned into the Control Plane Nodes and the Workers. One of the three Nodes is configured as a Control Plane Node and the rest as Worker nodes, following the master-slave pattern, that is commonly used in distributed architectures. A Control Plane with a single Node in Kubernetes means that there is a single point-of-failure for the orchestration of the internal cluster processes. In case we intended to emphasize High Availability (HA) even more, we would add more nodes in the control plane, and usually for that we need a minimum of 5 VMs. In our case, we chose only one master Node due to a lack of resources [44].

4.2 Kubernetes Setup

There are several approaches to creating and bootstraping the Kubernetes cluster. We set up the Kubernetes cluster using the Kubeadm software tool. It is a tool designed and built to provide the necessary commands to bootstrap a minimum viable Kubernetes cluster. The prerequisites for the

cluster setup are that each VM has the minimum amount of resources as stated in the Kubernetes documentation [45], that all nodes can communicate with each other, and that there is a container runtime installed in each machine (we use containerd, which is preferred).

Kubernetes components use specific ports for internal node communication. When setting up the cluster on premises we should be aware of the used ports [46] and open them with the appropriate firewall updates (in our case we used `iptables` to update the firewall rules).

We first configure the control plane nodes and the worker nodes and bootstrap them together.

In order for the multiple nodes to communicate with each other we need to install a CNI (Container Network Interface), which will handle all the virtual IPs given to each Pod along with the internal communication of the nodes. In our setup, we use the Calico CNI as one of the most popular and well-maintained ones.

In order to monitor the cluster for resource consumption and potential errors we also install a Kubernetes Metrics Server [47] with which we can get system information for all nodes through the masters' command line.
Find a more detailed description of the setup in the following article on Medium (here).

Worker Nodes

After setting up the Master Node, we perform the same initialization procedure for the Worker Nodes. For these Nodes, instead of using the `kubeadm init` command, we will use the `kubeadm join` command that the Master Node issues upon initialization. Running this command on the workers will add them to the cluster and we should see all nodes being in the 'Ready' state.

4.3 Dynamic Provisioner

Before we start deploying all the various components of our system, we must ensure that we have a Dynamic Provisioner installed in our cluster.

A dynamic provisioner is a service that has control over the storage of our machines and is able to create Volumes dynamically when a Pod requests it. These volumes are called Persistent Volumes as mentioned earlier, they have a fixed capacity and we want to avoid creating them manually to reduce human intervention.

If you are using a managed Kubernetes service (AWS EKS, GKE, AKS) the cloud provider will provide a StorageClass which will perform automati-

cally the PersistentVolume creation [14]. However, when using a bare-metal Kubernetes cluster, as we do in this example, we will need to manage storage by ourselves. At this moment, one of the open-source local storage provisioners is the one developed by rancher [48].

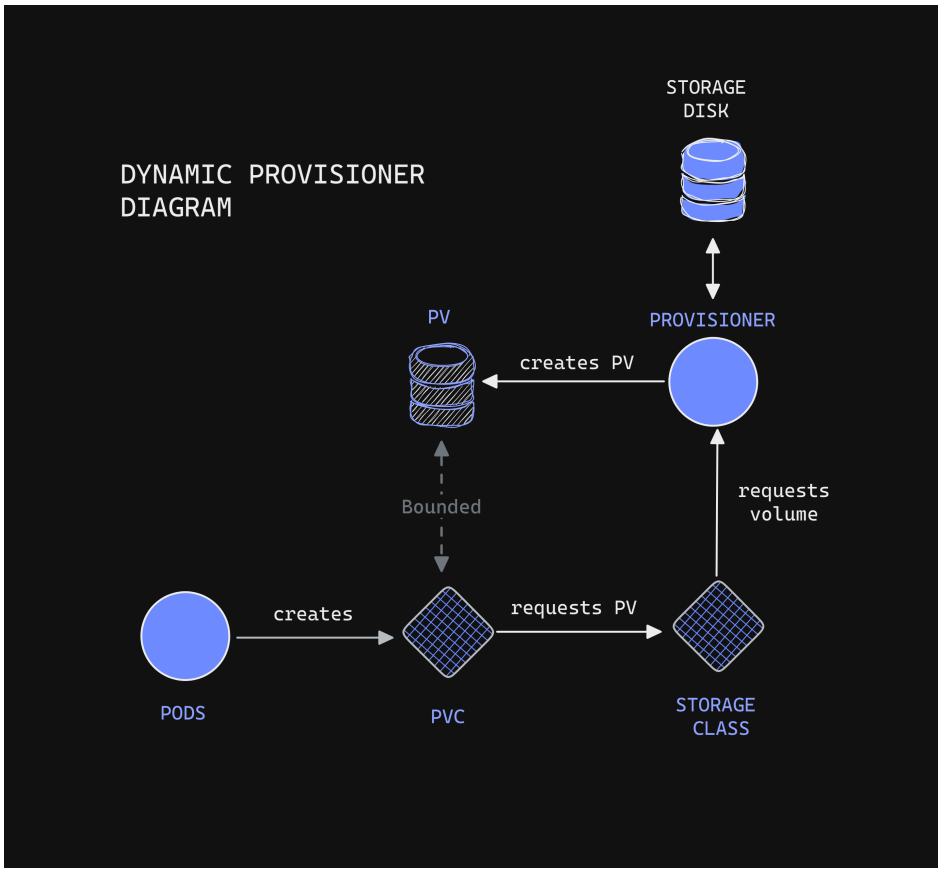

Figure 4.1: Diagram of dynamic PV creation

4.3.1 Deployment Instructions

Apply the manifest provided by rancher which deploys all the necessary CRDs for the provisioner to work.
`kubectl apply -f https://raw.githubusercontent.com/rancher/local-path-provisioner/v0.0.26/deploy/local-path-storage.yaml`
where the version `v0.0.26` should represent the latest tag of the provisioner releases [49].

The main components this manifest deploys, are the container of the Provisioner [50] with the equivalent tag deployed as a ReplicaSet, as well as

a StorageClass object named `local-path`. StorageClass is the object one must reference for the creation of the PersistentVolumes. After installation, you should see something like the following:

```
$ kubectl -n local-path-storage get pod
NAME                                     READY     STATUS    RESTARTS   AGE
local-path-provisioner-d744ccf98-xfcbk   1/1       Running   0          1m
```

Figure 4.2

4.3.2 Usage Instructions

To use the Dynamic Provisioner to create PersistentVolumes, one must reference the `local-path` StorageClass in the appropriate CRD. The most common object in Kubernetes that makes use of the StorageClass is the StatefulSet, which includes the `storageClassName` as a property field. An example of how one can reference the StorageClass to create PVs dynamically is shown below:

```
volumeClaimTemplates:
  - metadata:
      name: pvc
    spec:
      storageClassName: "local-path"
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: "1Gi"
```

It is important to note that if the `storageClassName` field is excluded then the 'default' StorageClass will be used and in the case where we set an empty string `storageClassName: ""` then no provider will be used and it is implied that a Persistent Volume already exists or will be created manually.

4.4 MQTT Broker (HiveMQ)

To deploy our MQTT Cluster of HiveMQ Brokers we will use the Operator provided by HiveMQ [51].

The Operator is deployed as a Helm Chart. The process we follow in this circumstance is to download the files locally, so that we have better control of what we deploy in our system, and then change the properties necessary in the `values.yaml` file to fit our system needs (for example the number of instances or the tag of the container).

4.4.1 Deployment Instructions

Clone the files from GitHub repository of the Helm Chart to your local file system.
`git clone https://github.com/hivemq/helm-charts.git` and navigate to the `/helm-charts/charts/hivemq-operator` folder.

Then, make sure you install any other dependency charts with the `helm dependency build` command.

Edit the `values.yaml` file and configure all the appropriate properties.
We edit the following properties:

- `nodeCount` → number of brokers
- `cpu` → CPU requirements of Node
- `memory` → memory requirements of Node
- Enable the `LoadBalancer` Service by uncommenting the line 272 *Because the MQTT messages will not get filtered by the Ingress Controller and we have to use another way, such as a LoadBalancer*

After that, we install the Helm Chart with the name `hivemq` , using the following command `helm install hivemq .`

The operator pod is initialized and after a few minutes, the MQTT Broker pods have been created, as many as the nodeCount property that we specified earlier.

4.4.2 Usage Instructions

We access the MQTT brokers through the IP of any of the Nodes or their DNS name because we have a LoadBalancer. We use port 1883, which is the default port for the MQTT protocol. In case someone wants to use a different port, they can change it from the `values.yaml` configuration file.

4.5 Messaging Broker (Apache Kafka)

For the deployment of the Kafka Brokers, we will use a simple way for Deploying it. We avoided using an operator or a Helm Chart, just because a Kafka cluster is not ideal for scaling up or down. Therefore, we need to choose a specific number of brokers from the beginning, depending on the expected load, and deploy them using a simple StatefulSet to demonstrate how simple their configuration can be.

The most common dependency of Kafka is ZooKeeper [27], but we deploy it with KRaft which does not require extra Pods to be deployed and is declared as a production-ready service [52]. One of the tasks of Kraft is the leader election among the different Kafka Brokers [53].

Below we have the deployment architecture of the Kafka Brokers deployed as a StatefulSet in Kubernetes.

4.5.1 Deployment Architecture

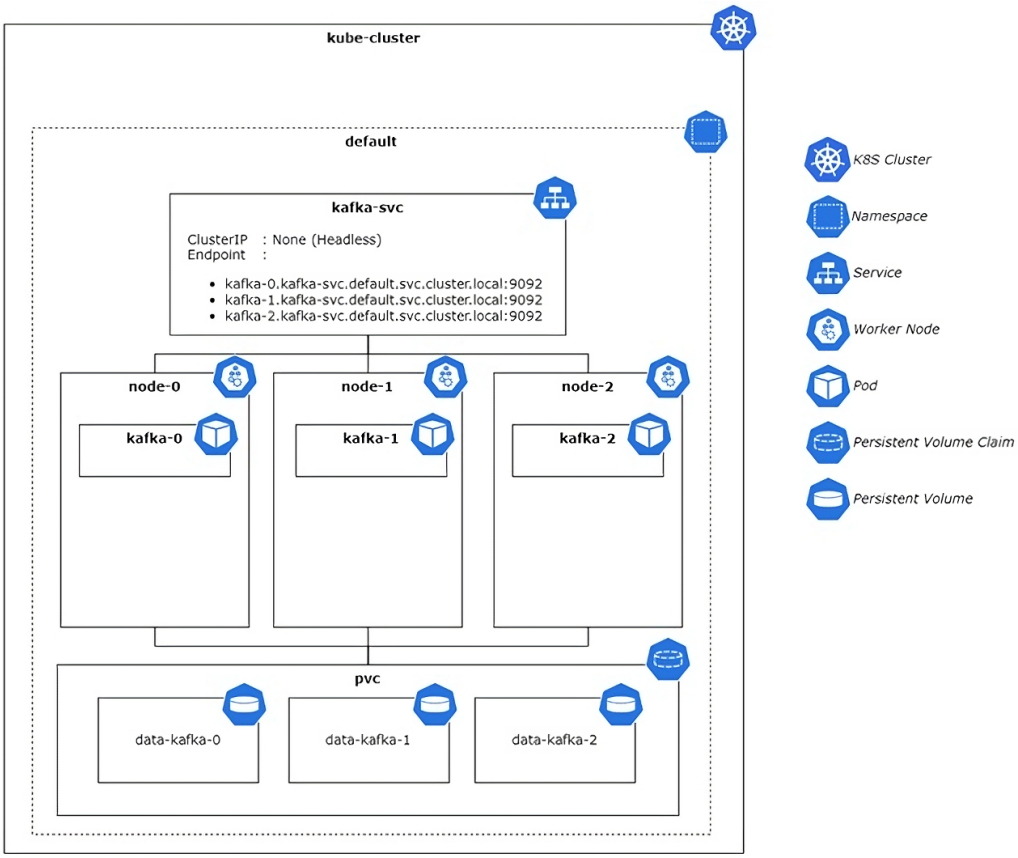

Figure 4.3: Kubernetes deployment architecture for Kafka [54]

4.5.2 Deployment Instructions

Our deployment process is based on the method used in the following article [54]. We deploy a StatefulSet, along with a Headless Service [55] to expose the Pods.

Here are the manifests for the StatefulSet and the Headless Service. We declare the number of Brokers we want, along with the StorageClass we deployed earlier.

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: kafka
  labels:
    app: kafka
spec:
  serviceName: kafka-svc
  replicas: 2
  selector:
    matchLabels:
      app: kafka
  template:
    metadata:
      labels:
        app: kafka
    spec:
      containers:
        - name: kafka-container
          image: doughgle/kafka-kraft:latest
          ports:
            - containerPort: 9092
            - containerPort: 9093
          env:
            - name: REPLICAS
              value: '2'
            - name: SERVICE
              value: kafka-svc
            - name: NAMESPACE
              value: default
            - name: SHARE_DIR
              value: /var/run/kafka-data
            - name: CLUSTER_ID
              value: LRx92c9yQKws786HYosuBn
            - name: DEFAULT_REPLICATION_FACTOR
              value: '2'
            - name: DEFAULT_MIN_INSYNC_REPLICAS
              value: '1'
          volumeMounts:
            - name: pvc
              mountPath: /var/run/kafka-data
...
```

```
...
  volumeClaimTemplates:
    - metadata:
        name: pvc
      spec:
        storageClassName: "local-path"
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: "1Gi"
```

```
apiVersion: v1
kind: Service
metadata:
  name: kafka-svc
  labels:
    app: kafka
spec:
  clusterIP: None # headless service
  ports:
    - name: '9092'
      port: 9092
      protocol: TCP
      targetPort: 9092
  selector:
    app: kafka
```

To deploy them together we will make use of Kustomize [56], a tool in Kubernetes which allows for multiple manifest deployments from a single file.

We create therefore, a `kustomization.yaml` file where we include the names of the StatefulSet and the Service manifests.

```
resources:
  - kafka-statefulset.yaml
  - kafka-service.yaml
```

After we create the kustomization file we apply it using the `kubectl` tool with the following command.
`kubectl apply -k .` where `.` is the current folder where we have

placed all the manifests along with the kustomization file.

After the deployment has been completed we should be able to see the Kafka Pods and PersistentVolumes having been created.

4.5.3 Usage Instructions

To connect with the Kafka Brokers, one must declare the names of all the brokers, along with their external ports. Because of the StatefulSet properties, the pods will have standardized names depending on the number of replicas we have. Therefore, we connect to the Brokers using their DNS name in Kubernetes and port 9092 (which is a default for Kafka) like the following.

Kafka Broker 1: `kafka-0.kafka-svc:9092`
Kafka Broker 2: `kafka-1.kafka-svc:9092`
Kafka Broker 3: `kafka-2.kafka-svc:9092`

where `kafka-svc` is just the name of the service we deployed earlier.

4.6 MQTT - KAFKA Connector

We deploy our custom MQTT to Kafka Connector with a Deployment object.

We note that the connector uses the previously deployed Services of MQTT (LoadBalancer) and Kafka (Headless Service - ClusterIP) as endpoints to connect to the appropriate brokers and transfer messages between them. All the details of the load-balancing of the messages and the connection to the Brokers are described in the 'Design & Concepts' section.

4.6.1 Deployment Instructions

The parameters we must configure are the following:

- The number of replicas we want
- The name of the Deployment (depending on the topic of messages we want to forward)
- The topic of the messages with the format we described earlier (using the $share keyword)

After that, we simply deploy our ReplicaSet using the following manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mqtt-kafka-connector
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mqtt-kafka-connector
  template:
    metadata:
      labels:
        app: mqtt-kafka-connector
    spec:
      containers:
        - name: mqtt-kafka-connector
          image: nickbel7/mqtt-kafka-connector:latest
          env:
            - name: MQTT_PORT
              value: '1883'
            - name: MQTT_TOPIC
              value: '$share/test-group/test-topic'
```

We should see the pods getting created and also be able to consume the messages that get sent to the MQTT Broker from the Kafka Brokers. Now we are ready to distribute these messages to all the rest of the components in our system through the Kafka Brokers.

To do that we need to make use of the Kafka Connect component, on top of which we will be able to deploy connectors for the various databases we will use.

4.7 Kafka Connect (with Timescale connector)

For the deployment of the Kafka Connect Brokers cluster we will make use of Helm Charts.

4.7.1 Deployment Instructions

We clone the files from GitHub repository of the Helm Chart to your local file system.

```
git clone https://github.com/confluentinc/cp-helm-charts.git
```

and navigate to the `/cp-helm-charts/charts/cp-kafka-connect` folder.

Edit the `values.yaml` file and configure all the appropriate properties.
We edit the following properties:

- `replicaCount` → number of instances
- `replication.factor` → how many replicas for data (≤ than the Kafka brokers)
- `kafka.boostrapServers` → the Kafka brokers URLs (comma seperated)

After that, we install the Helm Chart with the name `kafka-connect` , using the following command `helm install kafka-connect .`

The Kafka-Connect pods then are deployed and listen to port 8083 for the deployment of Connectors. However, these connectors are dependent on specific libraries that are not included in the simple deployment of the Kafka Connect cluster. Therefore, we must update the init script of the containers, which is the script that the container runs before it starts, to include the download of the appropriate libraries for the connectors we want to deploy.

Apart from that, we must also include in the init script the automatic deployment of the connector, which is a simple HTTP request, because in the case where the pods go down, then when recreated the connectors will be lost. Kafka-connect is not dependent on a Persistent Volume and therefore is not considered a stateful application in Kubernetes.

We present how we modify the init script of the containers to deploy automatically a connector for the Timescale database, which is just like any regular Postgresql database.

Inside the folder :
`/cp-helm-charts/charts/cp-kafka-connect/templates` we find the file `deployment.yaml` which includes the `command` property for the kafka-connect container. In this part of the manifest, we install our libraries, as follows.

```
command:
  - /bin/bash
  - -c
  - echo "======>␣Importing␣confluent-hub␣jars" &&
    confluent-hub install confluentinc/kafka-connect-jdbc
        :10.7.4 --component-dir /usr/share/java --no-prompt
        &&
    echo "Launching␣Kafka␣Connect␣workers" &&
    /etc/confluent/docker/run &
    $CREATE_CONNECTORS_SCRIPT &
    sleep infinity
```

After installing the JDBC connector, we then go and create the connector through the `$CREATE_CONNECTORS_SCRIPT` that we call, in the above script.

This variable represents a ConfigMap object [57], that we have previously deployed containing the properties of the connector.

What we did is create a `create-connectors.sh` file containing the script for creating the connector through a HTTP request and then creating a ConfigMap that references that file, as follows:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ template "cp-kafka-connect.fullname" . }}-
      connectors-configmap
  labels:
    app: {{ template "cp-kafka-connect.name" . }}
    chart: {{ template "cp-kafka-connect.chart" . }}
    release: {{ .Release.Name }}
    heritage: {{ .Release.Service }}
data:
  create-connectors.sh: |-
{{ .Files.Get "create-connectors.sh" | indent 4 }}
```

As a result, every time the instances of Kafka Connect initialize, they download the necessary libraries and also create the connectors to be able to process messages immediately.

4.8 TimescaleDB

For the deployment of the TimescaleDB cluster we will make use of Helm Charts.

4.8.1 Deployment Instructions

We clone the files from GitHub repository of the Helm Chart to your local file system.
`git clone https://github.com/timescale/helm-charts.git`
and navigate to the `/helm-charts/charts/tiemscaledb-single/` folder.

Then, make sure we install any other dependency charts with the `helm dependency build` command.

Edit the `values.yaml` file and configure all the appropriate properties.
We edit the following properties:

- `replicaCount` → number of instances
- `credentials.PATRONI_admin_PASSWORD` → admin password
- `storageClass` → edited to be 'local-path' as deployed earlier
- `persistenVolumes.data.size` → size of PersistentVolume
- `image.tag` → changed to pg15.4-ts2.12.2 (due to a bug in patroni)

After that, we install the Helm Chart with the name `timescaledb` , using the following command: `helm install timescaledb .`

4.8.2 Usage instructions

We should be able to access our database now through its DNS name `timescaledb.default.svc.cluster.local` and by using the admin code we specified in the `values.yaml` file earlier.

All the commands inside the database are the same as any Postgresql database. Timescale does not come with a User Interface, therefore we will use Grafana to monitor and visualize the data.

4.9 Cert-manager

When exposing our services to the outside world through an HTTP endpoint we want certificates to automatically get generated and renewed for it, thus, providing HTTPS connections. For that use, we deploy cert-manager, which creates TLS certificates for workloads in Kubernetes and renews them before they expire [58]. Along with cert-manager we need a certificate authority which will be the one that issues the certificates. Among the different alternatives, we will choose lets-encrypt.

For the deployment of the cert-manager pods we will make use of Helm Charts.

4.9.1 Deployment Instructions

Clone the files from GitHub repository of the Helm Chart to your local file system.
`git clone https://github.com/timescale/helm-charts.git`
and navigate to the `deploy/charts/cert-manager` folder.

Then, make sure you install all the necessary CRDs for the deployment to work.

```
kubectl apply -f https://github.com/cert-manager
/cert-manager/releases/download/v1.13.3
/cert-manager.crds.yaml
```

Edit the values.yaml file and configure all the appropriate properties.
We edit the following properties:

- `image.tag` (in all places) → get the latest version from here

After that, we install the Helm Chart with the name cert-manager , using the following command
`helm install cert-manager .`

4.9.2 Usage Instructions

Cert-manager pods should now be running in the cluster.
For the certificates to get generated automatically we should deploy a ClusterIssuer.

We apply the following manifest:

```yaml
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    # The ACME server URL
    server: https://acme-v02.api.letsencrypt.org/directory
    # Email address used for ACME registration
    email: youremail@domain.com
    # Name of a secret used to store the ACME account
       private key
    privateKeySecretRef:
      name: letsencrypt-prod
    # Enable the HTTP-01 challenge provider
    solvers:
    - http01:
        ingress:
          class: nginx
```

We have stated that in our ClusterIssuer that:

- Our ClusterIssuer is named 'letsencrypt-prod'
- We use the letsencrypt server to generate the certificates
- We use our email to sign the certificates
- We generate certificates for an Ingress Controller named 'nginx'

We should now be able to generate automatically certificates for our HTTP endpoints. An example on how to do so in an Ingress rule is demonstrated in the following manifest.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: some-ingress-rule
  annotations:
    cert-manager.io/cluster-issuer: letsencrypt-prod
...
```

4.10 Rest of components (Grafana, Hasura, Keycloak, Frontend, Backend)

Following the same practices for stateful and stateless applications we deploy them with the help of a HELM Chart if there is one, or as a simple deployment in the use case of our proprietary Frontend and Backend applications.

We provide all the necessary repositories for the abovementioned open-source software:

- **Ingress Controller** - HELM Chart
- **Grafana** - HELM Chart
- **Hasura** - HELM Chart
- **Keycloak** - HELM Chart
  (needs a separate deployment of a postgresql database HELM Chart)
- **Frontend, Backend** - Deployment + Service

Chapter 5

Testing & Evaluation

In the Testing and Evaluation section of the thesis, we delve into the process of evaluating a Kubernetes-based system, focusing on the system's capacity to handle high enough message throughput without failure. This involves designing tests that assess how the system manages and processes messages under various loads, identifying the maximum throughput it can sustain while maintaining stability and performance. Through these evaluations, we aim to understand the system's limits and capabilities in handling real-world operational demands.

We should note that the experiments do not involve actual data from IoT sensors, but rather simulations of realistic message payloads sent to the MQTT endpoint of the cluster just for a few seconds.
Also, during the testing, some of the components of the system, such as the frontend application were temporarily disabled due to the fact that they were not essential for the messaging pipeline, as well as for saving up some of the system's confined resources.

5.1 Experimental Setup

The experimental setup for the proposed proof-of-concept system is presented below. As mentioned earlier, our Kubernetes (K8s) cluster was deployed on top of 3 VMs, 1 acting as the Master Node and the rest 2 as the Worker Nodes. They are all hosted on proprietary hardware that uses a hypervisor to share resources.

The latest and most stable versions were used wherever possible.
Kubernetes v1.28.5 was installed on all 3 machines, along with containerd v1.7.2 for running the containers.

The hardware characteristics of each VM are displayed below:

VM Name	OS	CPU	RAM	DISK
Master Node	Ubuntu 22.04.2 LTS	2	16 GB	1 TB
Worker Node 1	Ubuntu 22.04.2 LTS	2	16 GB	1 TB
Worker Node 2	Ubuntu 22.04.2 LTS	2	16 GB	1 TB

Table 5.1: System Specifications

5.2 Performance Analysis

The goal of the performance tests for the system is to evaluate how much load it can handle, given very few resources and replicas for each component. We remind you that the purpose of the system is to use software that can run in a distributed way for each component. Thus, having 2 instances for each component is the minimum amount possible, without violating the initial constraint.

We want to prove that the system can tolerate enough message throughput that represent a big enough amount of actual home devices. By finding the maximum amount of devices that can connect to this system, we can define how to scale it accordingly to handle bigger magnitudes of IoT devices. Theoretically, no components will need to be replaced for bigger workloads, as all are designed to scale horizontally.

5.2.1 Message Throughput

In order to measure the throughput of the subsystem MQTT-to-Kafka end to end we performed Performance (Load) Testing. For each component of the subsystem (MQTT Brokers, MQTT-Kafka Connector, Kafka Brokers) we deployed the minimum possible amount of instances, which is 2. For those 2 instances of each component, we want to measure the input throughput for the MQTT Brokers and the output throughput for the Kafka Brokers.

For performing such a task, we concluded on using the Apache JMeter for the one end, [59] which can perform many MQTT requests into a small time window (with the MQTT plugin by EMQX [60]. It offers the option to perform requests with multiple clients and multiple threads, to better assess the limits of the tested system.

For the other end, which is the reception of the messages from Kafka, we created a custom container, that listens to the appropriate topic and measures the throughput of the messages by the second. We should note

here, that this measurement is not distributed and is performed by only one thread, thus, it might not represent the performance of the actual system. For such a small scale, however, it is adequate and proves our initial theory.

We tried a few different configurations in terms of the number of messages that were received and the number of threads that were used during the MQTT messages production. Below we present the results for the system for our various tests.

Test 1

Table 5.2: Publisher (MQTT)

Threads	**Total messages**	**Runtime (s)**	**Throughput (msg/s)**	**Throughput (bytes/s)**
1	1577	5	312 msg/s	78 kB/s

Table 5.3: Subscriber (Kafka)

Total messages	**Throughput (msg/s)**
1000	296 msg/s

Test 2

Table 5.4: Publisher (MQTT)

Threads	**Total messages**	**Runtime (s)**	**Throughput (msg/s)**	**Throughput (bytes/s)**
2	3660	5	422 msg/s	106 kB/s

Table 5.5: Subscriber (Kafka)

Total messages	**Throughput (msg/s)**
1000	357 msg/s

Test 3

Table 5.6: Publisher (MQTT)

Threads	**Total messages**	**Runtime (s)**	**Throughput (msg/s)**	**Throughput (bytes/s)**
5	4102	5	377 msg/s	95 kB/s

Table 5.7: Subscriber (Kafka)

Total messages	**Throughput (msg/s)**
1000	336 msg/s

Test 4

Table 5.8: Publisher (MQTT)

Threads	**Total messages**	**Runtime (s)**	**Throughput (msg/s)**	**Throughput (bytes/s)**
2	3678	5	414 msg/s	103 kB/s

Table 5.9: Subscriber (Kafka)

Total messages	**Throughput (msg/s)**
2000	370 msg/s

Test 5

Table 5.10: Publisher (MQTT)

Threads	**Total messages**	**Runtime (s)**	**Throughput (msg/s)**	**Throughput (bytes/s)**
1	2061	5	411 msg/s	103 kB/s

Table 5.11: Subscriber (Kafka)

Total messages	**Throughput (msg/s)**
1000	365 msg/s

Because all tests had very similar results, graphically speaking, we present the results for the last of the tests, which had a throughput of around 365 messages per second. Also, we present the diagram of the progressive transmission of the MQTT messages by JMeter.

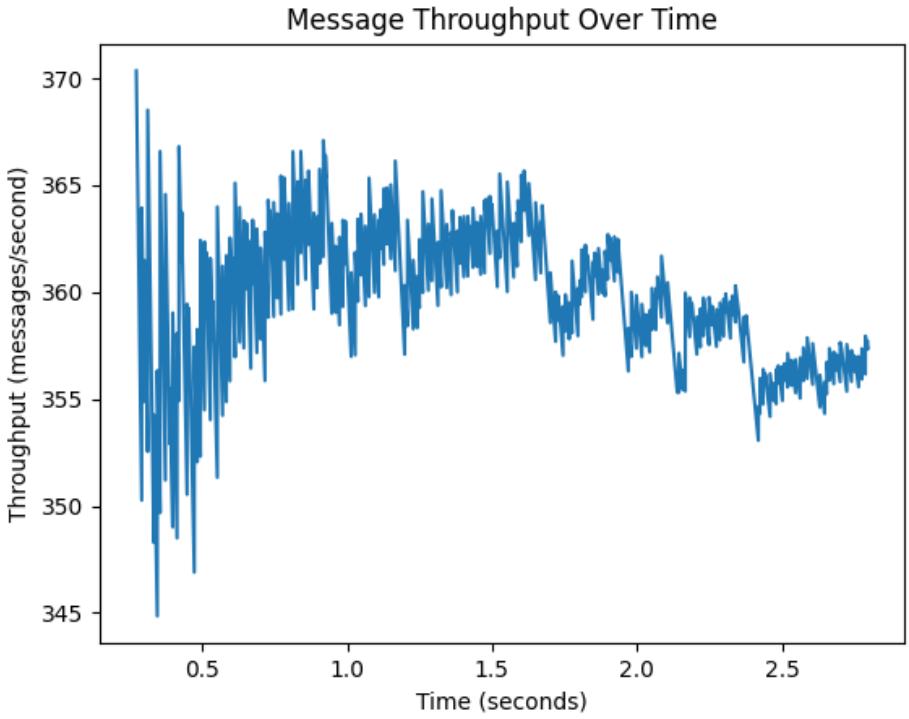

Figure 5.1: Message throughput of Kafka Subscriber

5.2.2 Results

We can see that the input throughput is not analogous to the number of threads that we used for the MQTT message producer. For many threads (5 or more), the MQTT Brokers allow as many messages to be processed simultaneously as for fewer threads. This implies that for just 2 MQTT Brokers there is a throughput limit of around 450 messages per second or around 110 kB per second that can be processed (this number is subject to change with future tests).

Another thing that we can observe is that the throughput of the Kafka Consumers is not lagging a lot from the input throughput, suggesting that it can scale along with the number of the MQTT Brokers. The difference in MQTT Publisher and Kafka Subscriber throughput is explained by the pre-processing of messages by the MQTT-Kafka Connector, which is responsible for adding a timestamp and transposing the message before it forwards it to the Kafka Brokers.

We conclude that the system can handle (from the Kafka endpoint) around 370 messages per second (large messages containing the schema).

This translates to either 370 devices transmitting every second or to around 22000 devices transmitting every minute. Because the application of this system is the processing of smart home energy consumption devices, it is interesting to note that by taking the median of 5 devices per home, the system can support around 4.400 homes, with the minimum amount of replicas for all components of the system.

5.3 Resource Management

When we have such a system, which is deployed on-premises and is an experimental work, we want to have as much control and information about the system as possible. Such information can give us real-time insight into the resource usage for every Node and individual component of the system, enabling us to prevent errors related to insufficient resources and plan accordingly for future expansions of the system. Also, we would be able to pinpoint any overuse of the available resources and optimize for less resource consumption and therefore lower costs.

In Kubernetes, the resource usage metrics, such as CPU and Memory usage, are available through the Metrics API [61]. It provides resource real-time usage information about the Pods and the Nodes. The only problem is that these metrics are only saved in memory and are not accessible for a long period. For that, we would need a separate database. 'Metric Server' is deployed as a 'Deployment' and acts as a cluster-level component that gets all these metrics by talking to the Kubelet through the Summary API. To use the 'Metric Server' we deployed it separately from the K8s installation, as presented earlier, and accessed it with the `kubectl top` command.

Results that were observed through this tool are shown and discussed as follows:

5.3.1 Required resources

We focused on observing the resource consumption from the Pods, which represent the deployed components in our system. We chose to track the main components of our message processing pipeline, firstly because these are the ones that we are concerned with in this work and secondarily because for these we have more control over their parametrization to make them run more efficiently.

We have measured the consumption of CPU and Memory utilization for two discrete states of the Pods. The first is what we call the 'idle' state where the Pods have just been deployed and they do not process or exchange

any messages (this represents the minimum amount of resources they need to operate). The second one is the state where we have successfully transmitted thousands of MQTT messages and the Pods are in the middle of processing them. With this second measurement, we plan on investigating any sharp differences in comparison to the 'idle' state.

Results are shown in the table below. Wherever we have multiple Pods for the same component we calculate the average value for each metric.

Table 5.12: System Resource Usage

Pod	**CPU (idle)**	**Memory (idle)**	**CPU**	**Memory**
MQTT Brokers (average)	15m	400Mi	35m	525Mi
MQTT-Kafka Connectors (average)	2m	10Mi	5m	14Mi
Kafka Brokers (average)	22m	1000Mi	25m	1050Mi
Kafka Connect (average)	10m	800Mi	17m	870Mi
Timescale DB (average)	5m	126Mi	10m	130Mi
Postgresql DB	6m	25Mi	6m	25Mi
Ingress Controller	2m	78Mi	2m	78Mi
Grafana	21m	760Mi	60m	960Mi

Based on the results, we observe that the amount of memory most of the components use is significant even in the 'idle' state. The CPU utilization is not ameliorable, in both states, but does not pose any pressure to the system, as the sum of it represents a relatively small percentage of the available system's CPU compute. Memory consumption, however, scales fast, with components like the Kafka Brokers and Kafka Connect growing very close to or even surpassing the 1Gb threshold of memory usage. It becomes evident, that if we want to deploy a lot of components into our system, we either have to provide a lot of memory to our worker Nodes or deploy smaller instances of the components.

In the case now, where we want to have very few components, but want them to handle very large loads, this initial test proves that the change in resource consumption is not significant enough to suggest an extension of the available memory.

If there is an adequate amount of available memory then the system will theoretically hold and therefore we can focus on deploying more replicas, which will lead to more threads processing information at the same time and a faster system

5.3.2 Resource optimization

In practice, containers have no upper bound on their CPU and memory usage. They can consume all the resources that are available on the node, where it is running on. In the case of memory, this might invoke the 'Out of Memory (OOM)' killer and there is a possibility for the container to be killed. To avoid this, Kubernetes gives us the possibility to manage the amount of resources that would be consumed by containers. By configuring resource limits for the containers running on our cluster, we avoid losing jobs. Moreover, this way, we can make efficient use of the available resources on our cluster's nodes. We define such resource constraints in the Pod definition, with two sub-categories: 'requests' and 'limits'. In 'requests', we define a reasonable value for the memory and CPU requests of the Pod to be running properly. Defining this value is a help for scheduling of Pods in the nodes with appropriate available resources. Then, in the 'limits' field, we define the upper bound that the Pod can use. Hence, the containers would not be allowed to use the whole of available resources on the node [9].

5.4 Network Bandwidth

To analyze most of the system's components and identify current and future bottlenecks, we tested the network bandwidth between the Kubernetes cluster nodes to ensure it does not pose any limits.

We used the `iperf` tool on the nodes to measure the bitrate between them. We found that there is an actual connection of 1GB/s, which is more than enough and does not pose any limit to the messages exchanged between the nodes.

The results of the networking test are shown below :

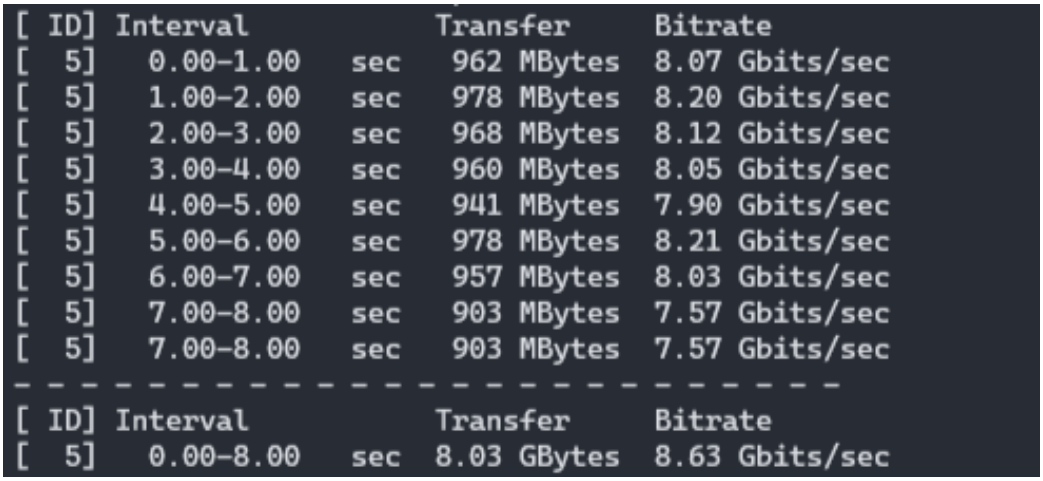

Figure 5.2: Results from network bandwidth testing

Chapter 6

Conclusion & Future work

In this thesis, the experimental design and evaluation of a Kubernetes-based IoT system reveal the ability for everyone to deploy it using open-source technologies that work in a distributed manner end to end across the pipeline. These technologies have matured enough to be stable for production purposes, enabling more and more people to use them and experiment with them, without relying solely on proprietary software from cloud providers.

Such findings underscore the potential of Kubernetes and associated technologies for deploying scalable, resilient IoT infrastructures, paving the way for future expansions and optimizations to accommodate growing operational demands.

At the same time that these distributed and scalable technologies become more stable and accessible, the IoT devices used for energy consumption increase at a fast pace due to our efforts to mitigate the energy inefficiency problem in big cities and power-hungry installations. IoT systems are set to become as relevant as ever for city-wide clusters of sensors to be able to function properly, but also for vast amounts of data to be collected from them and used for machine learning models.

6.1 Conclusions

The findings of this work are summarized below, through the answering of the initially stated Research Questions.

6.1.1 Research Question 1

Is it possible to use open-source production-level software that can run in a distributed manner for all components of such a system ?

After the deployment of the designed system we proposed earlier, along with the comparison between various technologies for each component, we conclude that an IoT system can be designed to function in a Kubernetes cluster with as low as 3 Nodes (1 Master and 2 Workers) by using open-source software, for each of its core components, that can run in a distributed manner among the Worker Nodes, whether that software is stateless or stateful, meaning that it has to keep a state of its data permanently stored and replicated.

Most of the used software contributors already offer the ability to deploy it using a HELM Chart, which means that they have already considered it being used in a distributed network of Nodes, which is what Kubernetes represents.

Also, because of the Services in Kubernetes and the internal DNS service it provides, it becomes a lot easier to perform service discovery for all the different instances of the deployed applications, even if they change IPs. This enables fast and easy deployment of not only existing production-grade software made for Kubernetes but of custom-made components such as a Kafka Connector or an API that needs to perform load balancing.

However, apart from the messaging brokers, the databases, and the stateless applications, there are some applications such as a data lake, which stores big amounts of data, that might not be ideal to get deployed in a distributed manner, because of the overhead that would be created with the data replication and state synchronization. Of course, the number of replicas for each stateful component matters, because on some occasions it can compromise its performance by unnecessarily increasing its fault-tolerance.

All in all, we were able to prove that for the essential components of an IoT System based on Kubernetes, there are mature enough open-source solutions that can combine all the benefits of a containerized application in the cloud and a distributed system.

6.1.2 Research Question 2

How does the system perform while using the minimum amount of resources for its distributed components ?

First of all, we should note that Kubernetes, and especially K8s is a relatively heavy container orchestration framework. There are lighter solutions like k3s [62], but k8s is the one we used for this specific demonstration. The main reason is that it uses a lot of controllers to provision the state of not only the cluster itself but one of the deployed applications' too. We note that,

because we should consider the extra memory, CPU, and network utilization from the system itself, apart from the deployed components of the system.

With the main components of the messaging processing pipeline deployed, we load-tested the system to conclude the load it could handle given the memory we had available. With only 2 replicas for each component, the system managed to successfully process around 370 concurrent messages per second, which can translate to approximately 4000 homes transmitting every 1 minute, which is adequate for our pilot testing of the EHS system.

The messages that we were sending were relatively large compared to the actual ones that will get sent by the MQTT devices because we incorporated in them the schema that is needed for them to be inserted into our time series database. By deploying and utilizing a schema registry, we should be able to increase the throughput of the system even more.

The minimum amount of resources refers to the number of replicas for each deployed component, which are only 2, but are able to scale horizontally to tolerate higher message throughput. We discuss how we plan on upscaling the system later in the 'Future Work' section.

6.1.3 Research Question 3

What are the practical limitations of such a system ?

Such a Kubernetes-based system poses many advantages, such as the easy interconnectivity of the Nodes and Pods through the internal Networking and DNS services or the better life-cycle of the containerized applications which require minimal human intervention.

The purpose of this work, however, was to also exploit some of the limitations of such a system where all components work in a distributed manner, given the confined amount of computer resources that we were given and which cannot grow indefinitely as they do in the cloud.

The limitations of such a horizontally scalable system usually lie in the number of available resources. Given the fact that we build the system using software that can run in a distributed manner, we should be able to scale indefinitely given multiple nodes and multiple clusters.

However, we faced problems during our deployment, with the use of memory by some scheduled pods, which were using too much memory. The response of Kubernetes was to evict them due to 'Memory Pressure' [SOURCE - accessed 14-February-2024] in order to not cause disruptions in other Pods' functionality. Memory pressure was the main deficit, of the system with the number of Cluster Nodes being the second most important one, having 3

Nodes in total. It shouldn't be a good practice to schedule Pods on the Master Node, therefore, the two Worker Nodes were responsible for being able to handle all the load of the deployed components. While in many stateless applications, this is not a problem, in many cases a deployment needed at least three replicas to be considered fault-tolerant. Usually, this happens in stateful applications where there is a consensus protocol involved, like the Raft protocol.

Additionally, concerning the open-source software, we faced problems regarding some bugs as well as inadequate documentation which was essential in order to parameterize it. This is why we ended up using helm charts where it was possible because they offered an almost out-of-the-box solution for the deployment of most components and were easier to uninstall in case there was a bug or something did not work as expected.

Finally, moving forward, we should also note that the actual IoT devices will have different versions of mqtt with some providing a timestamp while others don't. So this is a challenging topic regarding an IoT system that has to handle messages of various formats and protocols.

6.2 Future work

Our work demonstrates both the idealization and deployment of a Kubernetes-based system for IoT applications using distributed solutions. It sets the foundations for a truly distributed and horizontally scalable system that can simulate those that are run by cloud providers and other closed-source developers. However, as we pointed out, there are still limitations in the proposed solution that have to be addressed while we move towards a deployment for a real-world scenario.

Hence, we suggest the following additions to our existing architecture:

6.2.1 Autoscaling

For automatic scaling of the pods, Kubernetes provides an auto-scaling mechanism by implementing a Horizontal Pod Autoscaler (HPA), which is implemented as a controller. There are many options and custom autoscalers, whose primary job is to upscale or downscale individual deployments according to some specified metrics, such as CPU utilization.

In the future, we plan on using the KEDA autoscaler in our system, primarily for parts of it that are observed to have sudden spikes of requests, such as the Frontend and Backend service.

KEDA [63] is a Kubernetes-based Event event-driven autoscaler that works alongside the HPA and extends its functionality. With it, we will be able to map the components that we want to scale in an event-driven way.

6.2.2 Monitoring metrics

Because of the complexity of the system in terms of the number of components, as well as the continuous flow of data, we need system metrics in order to monitor them and scale the system dynamically.

Prometheus [64] works as a time series database that can collect data from crucial components of the pipeline, such as the Kafka Brokers, and depict them in a Grafana graphical interface or use them to alert the administrators in case an error occurs.

6.2.3 CI/CD pipeline

Of course, we want the system to require as little human intervention as possible, not only for maintenance but for deployment of updated versions of its components too. That is why it is very important to utilize CI/CD tools that allow the application developers to deploy an updated version of their software without downtime of the system.

Such tools often come together with the repository manager in which we store our code. For example, GitHub offers GitHub Actions [65] and GitLab offers GitLab CI/CD [66].

6.2.4 Separate production and development pipelines

As in all traditional systems, we should offer to the developers that deploy their applications to our system the availability of both a Development and Production environment. In such a case, they will be able to deploy and test their code for any bugs on the Development workspace and if it functions as expected it should then be deployed in the Production workspace. More specifically, the way we intend to implement this is by using the namespaces in Kubernetes. Namespaces represent separated virtual environments inside the cluster, but without blocking the connection between the Pods. Therefore, we can deploy all the core components of the messaging pipeline in a common namespace (the 'default' namespace for example) and create a 'Development' and a 'Production' namespace for all the components that reside architecturally in the right most end of the system, such as the Frontend or the Keycloak.

6.2.5 Stress testing of each component separately

In our testing process, we only addressed the throughput of the messaging pipeline between the MQTT and the Kafka Brokers, because we only wanted to test for the bottlenecks in this specific part and measure its performance. However, moving forward, a production-grade system must have all its components individually tested, so that we can map future problems that occur with better precision, as well as focus on improving their individual performance. In such a complex and distributed system, if one component creates lag, then the rest will have decreased throughput too, just like a pipe system.

6.2.6 More Nodes

As good as it is to theorize about the scalability of the system, it is even better to test in practice how it scales when we add more nodes to our system. Right now, with 3 Nodes, our system is not very resilient, because even if 1 Node was taken down, then most distributed components would fail. In the worst case that the Master Node fails then the Cluster will be unable to heal pods or create new deployments. That is we should consider adding 2 more Nodes in our Kubernetes cluster, which will either get separated into a Control Plane Node and a Worker Node, or both will act as Worker Nodes, in case we want to emphasize increasing the maximum amount of messages the system can handle per second.

6.2.7 Ingress Controller as a Daemon set

In our effort to create such a highly distributed system, we should also emphasize in the balanced distribution of the networking components and load balancing of the requests.

In the simplified base system we developed, we used an Nginx Ingress Controller [SOURCE - accessed; 14-February-2024] as the reverse proxy that handles all the incoming requests that come into our Cluster. We deployed that controller as a single Deployment, meaning that we had a single Pod running in the Master Node that handles all the requests and distributes them accordingly. There is the option, however, to deploy the Ingress Controller as a DaemonSet, which in theory, should ease the load of the Master Node and make the system more performant, as multiple threads will handle the request distributions.

6.2.8 Extend device support to LoRa WAN

The initial purpose of this work was to create a base for an IoT system that should handle MQTT requests from energy consumption sensors installed inside homes. There is, though, the potential for this architecture to be used for the processing of different kinds of sensors and protocols, such as the LoRa WAN [67] devices, which can be installed in a wider field of range and are more suitable for smart city applications.

This adaptation would require the research of appropriate software that can handle these different packet types and data connectors that will format them as needed for the Kafka brokers to be able to process them.

6.2.9 Deploy a Data Lake

As mentioned earlier, part of the motivation for this work is not only to process and aggregate data, but also to store them in a very rich format, so that machine learning models can be trained on top of them.

For such a use case, we need to deploy a Data Lake, which is a type of Database that can store, process, and analyze vast amounts of data ignoring their data limits, in comparison to a standard SQL database which might struggle to handle them. Its main advantage is the scalability and ability to handle both unstructured and structured data (given a schema).

Bibliography

[1] *Internet evolution timeline*, https://blog.bytebytego.com/p/a-crash-course-in-networking, Accessed: 2024-02-16.

[2] B. Safaei, A. M. Hosseini Monazzah, M. Barzegar Bafroei, and A. Ejlali, "Reliability side-effects in internet of things application layer protocols", in *2017 IEEE International Conference on Smart Reliable Systems (ICSRS)*, IEEE, 2017, pp. 1–4.

[3] *Top iot development trends*, https://www.ibaseit.com/blog/top-iot-development-trends/, Accessed: 2024-01-25, 2023.

[4] *Iot: The future of web development*, https://www.knowledgehut.com/blog/web-development/iot-future, Accessed: 2024-01-25, 2023.

[5] D. Mlynka, "Iot device management using kubernetes", in *Diploma Thesis, Masaryk University, Faculty Of Informatics*, Adviser: RNDr. Zdeněk Matěj, Ph.D., Brno, 2022.

[6] *What is a Container?*, https://www.docker.com/resources/what-container/, Accessed: 2024-02-15.

[7] *Container orchestration tools*, https://www.redhat.com/en/topics/containers/what-is-container-orchestration, Accessed: 2024-02-15.

[8] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg", in *Proceedings of the Tenth European Conference on Computer Systems*, ACM, 2015, p. 18.

[9] M. Tavakkoli, "Analyzing the applicability of kubernetes for the deployment of an iot publish/subscribe system", in *Master's Thesis in Computer Science, EIT Digital Master's Programme in Cloud Computing and Services, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology*, Delft University of Technology and Nokia Bell Labs, Delft, The Netherlands, Oct. 2019.

[10] Kubernetes Authors, *Customresourcedefinitions*, https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources, Accessed: 2024-02-01.

[11] Kubernetes Authors, *Services - kubernetes documentation*, https://kubernetes.io/docs/concepts/services-networking/service/, Accessed: 2024-01-25.

[12] Kubernetes Authors, *Services in kubernetes - the service api*, https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types, Accessed: 2024-01-25.

[13] Kubernetes Authors, *Ingress - kubernetes documentation*, https://kubernetes.io/docs/concepts/services-networking/ingress/, Accessed: 2024-01-25.

[14] Kubernetes Authors, *Storage classes - kubernetes documentation*, https://kubernetes.io/docs/concepts/storage/storage-classes/, Accessed: 2024-01-30.

[15] Kubernetes Authors, *Controllers - kubernetes documentation*, https://kubernetes.io/docs/concepts/workloads/controllers/, Accessed: 2024-01-30.

[16] Kubernetes Authors, *Api concepts - kubernetes documentation*, https://kubernetes.io/docs/reference/using-api/api-concepts/, Accessed: 2024-01-30.

[17] Kubernetes Authors, *Extending kubernetes - kubernetes documentation*, https://kubernetes.io/docs/concepts/extend-kubernetes/, Accessed: 2024-01-30.

[18] Kubernetes Authors, *Operators - kubernetes documentation*, https://kubernetes.io/docs/concepts/extend-kubernetes/operator/, Accessed: 2024-01-30.

[19] M. Singh, *Consensus is a critical concept in distributed systems, where a group of nodes must work together*, https://medium.com/@mndpsngh21/consensus-is-a-critical-concept-in-distributed-systems-where-a-group-of-nodes-must-work-together-5c3b234df3b6, Accessed: 2024-01-29, 2023.

[20] *Hasura Documentation*, https://hasura.io/docs/latest/index/, Accessed: 2024-02-16.

[21] PsiBorg Technologies, *Healthcare solutions - psiborg*, https://psiborg.in/healthcare/, Accessed: 2024-02-15, 2024.

[22] B. Mishra and A. Kertesz, "The use of mqtt in m2m and iot systems: A survey", in *IEEE Access*, IEEE, vol. 8, 2020.

[23] *Mosquitto mqtt broker*, https://mosquitto.org/documentation/, Accessed: 2024-02-20.

[24] EMQ Technologies, *Emq x documentation*, https://www.emqx.io/docs/en/latest/, Accessed: 2024-01-25, 2024.

[25] VerneMQ, *Vernemq documentation*, https://docs.vernemq.com/, Accessed: 2024-01-25, 2024.

[26] *Apache kafka*, https://kafka.apache.org/, Accessed: 2024-02-20.

[27] Apache ZooKeeper, *Apache zookeeper*, https://zookeeper.apache.org/, Accessed: 2024-02-8, 2024.

[28] *Mqtt shared subscriptions guide*, https://cedalo.com/blog/mqtt-shared-subscriptions-guide/, Accessed: 2024-01-25, 2024.

[29] *Kafka Connect*, https://docs.confluent.io/platform/current/connect/index.html, Accessed: 2024-02-24.

[30] *Kafka connect concepts*, https://docs.confluent.io/platform/current/connect/index.html#connect-concepts, Accessed: 2024-01-26.

[31] *Schema registry documentation*, https://docs.confluent.io/platform/current/schema-registry/index.html, Accessed: 2024-01-25.

[32] *Apache kafka schema registry course*, https://developer.confluent.io/courses/apache-kafka/schema-registry/, Accessed: 2024-01-25.

[33] *Kip-301: Schema inferencing for jsonconverter*, https://cwiki.apache.org/confluence/display/KAFKA/KIP-301%3A+Schema+Inferencing+for+JsonConverter, Accessed: 2024-02-15.

[34] *Streaming replication*, https://wiki.postgresql.org/wiki/Streaming_Replication, Accessed: 2024-01-29.

[35] *High availability timescaledb & postgresql with patroni*, https://www.timescale.com/blog/high-availability-timescaledb-postgresql-patroni-a4572264a831/, Accessed: 2024-01-29.

[36] *The raft consensus algorithm*, https://raft.github.io/?ref=timescale.com, Accessed: 2024-01-29.

[37] *Patroni documentation for kubernetes*, https://patroni.readthedocs.io/en/latest/kubernetes.html, Accessed: 2024-01-29.

[38] *GraphQL: A query language for your API*, https://graphql.org/, Accessed: 2024-02-17.

[39] *Introduction to graphql subscriptions*, https://hasura.io/learn/graphql/intro-graphql/graphql-subscriptions/, Accessed: 2024-02-16.

[40] Cilium, *Ebpf - the future of networking*, https://cilium.io/blog/2020/11/10/ebpf-future-of-networking/, Accessed: 2024-02-15, 2020.

[41] C. Pahl, "Containerization and the paas cloud", *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[42] *Helm documentation*, https://helm.sh/docs/topics/charts/, Accessed: 2024-02-01.

[43] CircleCI, *What is helm?*, https://circleci.com/blog/what-is-helm, Accessed: 2024-02-01, 2020.

[44] S. R. Nadaf and H. K. Krishnappa, "Deploying containerized applications in a kubernetes cluster running in a public cloud", in *International Journal of Advanced Science and Computer Applications*, UKInstitute, vol. 2, 2023, pp. 7–18.

[45] Kubernetes, *Creating a cluster with kubeadm*, https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/, Accessed: 2024-01-30, 2024.

[46] *Kubernetes Ports and Protocols*, https://kubernetes.io/docs/reference/networking/ports-and-protocols/, Accessed: 2024-02-16.

[47] *Kubernetes metrics server*, https://github.com/kubernetes-sigs/metrics-server, Accessed: 2024-02-20.

[48] Rancher, *Local Path Provisioner*, https://github.com/rancher/local-path-provisioner, Accessed: 2024-02-07, 2024.

[49] Rancher, *Local Path Provisioner Tags*, https://github.com/rancher/local-path-provisioner/tags, Accessed: 2024-02-07, 2024.

[50] Rancher, *Local Path Provisioner v0.0.26 Docker Image*, https://hub.docker.com/layers/rancher/local-path-provisioner/v0.0.26/images/sha256-9325057706239e408ed417b19356cd892ee67b046ee?context=explore, Accessed: 2024-02-07, 2024.

[51] HiveMQ, *HiveMQ Operator Helm Chart*, https://github.com/hivemq/helm-charts/tree/master/charts/hivemq-operator, Accessed: 2024-02-08, 2024.

[52] *Learn about kraft*, https://developer.confluent.io/learn/kraft/, Accessed on: 2024-02-08.

[53] *Kafka leader election*, https://levelup.gitconnected.com/kafka-leader-election-4e7dfad2aa18, Accessed on: 2024-02-08.

[54] *Deploying a multi-broker kafka cluster in kubernetes*, https://www.mitrais.com/news-updates/deploying-a-multi-broker-kafka-cluster-in-kubernetes/, Accessed on: 2024-02-08.

[55] *Headless services*, https://kubernetes.io/docs/concepts/services-networking/service/#headless-services, Accessed on: 2024-02-08.

[56] *Managing kubernetes objects using kustomize*, https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/, Accessed on: 2024-02-08.

[57] *Configmaps*, https://kubernetes.io/docs/concepts/configuration/configmap/, Accessed on: 2024-02-16.

[58] *Cert-manager docs*, https://cert-manager.io/docs/, Accessed: 2024-02-23.

[59] *Apache jmeter*, https://jmeter.apache.org/index.html, Accessed on: 2024-02-14.

[60] *Mqtt jmeter plugin*, https://github.com/emqx/mqtt-jmeter, Accessed on: 2024-02-14.

[61] *Resource metrics api*, https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/resource-metrics-api.md, Accessed: 17-February-2024.

[62] *K3s Lightweight Kubernetes*, https://k3s.io/, Accessed: 2024-02-014.

[63] *KEDA*, https://keda.sh/, Accessed: 2024-01-07.

[64] *Prometheus*, https://prometheus.io/, Accessed: 2024-01-14.

[65] *Build your CI/CD pipeline with GitHub Actions in four steps*, https://github.blog/2022-02-02-build-ci-cd-pipeline-github-actions-four-steps/, Accessed: 2024-01-14.

[66] *GitLab CI/CD*, https://docs.gitlab.com/ee/ci/, Accessed: 2024-01-14.

[67] *About LoRaWAN*, https://lora-alliance.org/about-lorawan/, Accessed: 2024-01-14.