UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Microservices in Internet of Things Infrastructures for Smart City Solutions

An Event-Driven Air Quality Monitoring Application Utilizing Apache Kafka and Cloud Methodologies

Diploma Thesis

Nikolaos Kallinikos Kolovos

Supervisor: Georgios Stamoulis

March 2024

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Microservices in Internet of Things Infrastructures for Smart City Solutions

An Event-Driven Air Quality Monitoring Application Utilizing Apache Kafka and Cloud Methodologies

Diploma Thesis

Nikolaos Kallinikos Kolovos

Supervisor: Georgios Stamoulis

March 2024

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Δημιουργία μικροϋπηρεσιών σε δομές Διαδικτύου των Πραγμάτων για λύσεις έξυπνης πόλης

Μια εφαρμογή παρακολούθησης της ατμοσφαιρικής ποιότητας αξιοποιώντας Apache Kafka και μεθοδολογίες Cloud

Διπλωματική Εργασία

Κολοβός Νικόλαος Καλλίνικος

Επιβλέπων/πουσα: Σταμούλης Γεώργιος

Μάρτιος 2024

Approved by the Examination Committee:

Supervisor **Georgios Stamoulis**
Professor, Department of Electrical and Computer Engineering, University of Thessaly

Member **Charilaos Akasiadis**
Post-Doctoral Researcher, Institute of Informatics and Telecommunications, NCSR 'Demokritos'

Member **Nestor Evmorfopoulos**
Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly

Acknowledgements

Σελίδα με ευχαριστίες του συγγραφέα (προαιρετικό)

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

«Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism».

The declarant

Nikolaos Kallinikos Kolovos

Diploma Thesis

Microservices in Internet of Things Infrastructures for Smart City Solutions

An Event-Driven Air Quality Monitoring Application Utilizing Apache Kafka and Cloud Methodologies

Nikolaos Kallinikos Kolovos

Abstract

In this thesis, we design and develop an event-driven Internet of Things (IoT) Air Quality monitoring application, that tracks $PM_{2.5}$ particles concentration levels, humidity and temperature. We utilize advanced cloud computing tools, and expand the SYNAISTHISI platform.

Environmental air pollution has been a big concern for a lot of people lately. Our goal is to provide a user-friendly solution for anyone willing to monitor air quality in a given region. Any citizen should be informed of their local air quality, and beyond that, should be able to create their own tools to monitor it in an easy and accessible way. Starting with this basic principle, we utilize new industry standard cloud tools and services, in combination with existing Internet of Things protocol standards, creating a universally available and highly scalable application.

In recent years, there have been a lot of attempts to create simple affordable Internet of Things solutions for a plethora of problems -in our case air pollution- by using simple and accessible implementations. In our approach, we attempt to go a step further and implement a more complex in its development, but quite straight forward on its operation and capabilities, cloud based solution. In line with the principles of most IoT applications, we have designed our solution to be both accessible and user-friendly.

Additionally, we extend an existing research-oriented IoT platform by integrating the Apache Kafka protocol. Apache Kafka provides high throughput, low latency, and high availability capabilities to our application. The structure of our stack revolves around evolving the capabilities of our SYNAISTHISI platform, by utilizing new cloud tools that can provide backward compatibility with other protocols -MQTT, RabbitMQ-, that are currently widely

used by IoT edge devices and merge them with the advanced capabilities of Apache Kafka and its counterparts.

Our focus is on environmental measurements and distributing this data to any system that the SYNAISTHISI platform operators wish to utilize. The addition of Apache Kafka and its accompanying tools expands the potential for a wide variety of cloud computing applications. Thus, by developing our air quality application, we are able to explore the technologies that are utilized in our platform. These technologies can be implemented in various services and IoT applications, aligning with the primary vision of the SYNAISTHISI platform.

Keywords:

Internet of Things, $PM_{2.5}$ particles, cloud tools, SYNAISTHISI platform,air pollution, air quality, Apache Kafka

Διπλωματική Εργασία

Δημιουργία μικροϋπηρεσιών σε δομές Διαδικτύου των Πραγμάτων για λύσεις έξυπνης πόλης

Μια εφαρμογή παρακολούθησης της ατμοσφαιρικής ποιότητας αξιοποιώντας Apache Kafka και μεθοδολογίες Cloud

Κολοβός Νικόλαος Καλλίνικος

Περίληψη

Σε αυτή την εργασία αναπτύσσουμε μια σύνθετη εφαρμογή για το Διαδίκτυο των Πραγμάτων(IoT), όπου καταγράφουμε τα επίπεδα συγκέντρωσης $PM_{2.5}$ σωματιδίων, την υγρασία και τη θερμοκρασία. Η λύση μας ,αξιοποιεί αναπτυγμένες εργαλεία cloud και επεκτείνει την πλατφόρμα «SYNAISTHISI».

Η ατμοσφαιρική ρύπανση αποτελεί μια σημαντική ανησυχία για πολλούς ανθρώπους τα τελευταία χρόνια. Σκοπός μας είναι να προσφέρουμε μια λύση φιλική προς τον χρήστη και για όποιον επιθυμεί να παρακολουθεί την ατμοσφαιρική ποιότητα σε κάποια περιοχή. Κάθε πολίτης θα πρέπει να έχει το δικαίωμα να γνωρίζει την ποιότητα της ατμόσφαιρας όπου διαμένει και επιπλέον να είναι σε θέση να δημιουργήσει ακόμα και τα δικά του εργαλεία καταγραφής με εύκολο και προσιτό τρόπο. Βασισμένοι σε αυτό, δημιουργήσαμε μια πλήρως αξιοποιήσιμη και επεκτάσιμη εφαρμογή, αξιοποιώντας αναπτυγμένα cloud εργαλεία και τεχνολογίες, σε συνδυασμό με εγκαθιδρυμένα πρωτόκολλα του Διαδικτύου των Πραγμάτων(IoT).

Τα τελευταία χρόνια, έχουν υπάρξει πολλές προσπάθειες στη δημιουργία εφαρμογών του Διαδικτύου των Πραγμάτων, που βοηθούν στην επίλυση πολλών προβλημάτων (στην περίπτωση μας ατμοσφαιρικη ρύπανση), χρησιμοποιώντας απλές και προσιτές υλοποιήσεις. Αυτή τη φορά προσπαθούμε να πάμε ένα βήμα παρακάτω και να αναπτύξουμε μια σύνθετη στη δημιουργία της, αλλά ξεκάθαρη κατά τη χρήση της, cloud εφαρμογή. Ακολούθουμε τις βασικές αρχές των περισσότερων εφαρμογών του Διαδικτύου των Πραγμάτων, κάνοντας την υλοποίηση μας προσιτή και βολική προς του χρήστες της.

Το βασικό εργαλείο που αξιοποιούμαι και αναπτύσσουμε είναι το Apache Kafka, ένα ισχυρό εργαλείο που προσφέρει υψηλή ταχύτητα επεξεργασίας, χαμηλή απόκριση και αυξημένη διαθεσιμότητα στην εφαρμογή μας. Η δομή της υλοποίησης σκοπεύει στην επέκταση

της υπάρχουσας πλατφόρμας «SYNAISTHISI», αξιοποιώντας νέα εργαλεία τα οποία μας επιτρέπουν να διασυνδέσουμε υπάρχοντα πρωτόκολλα επικοινωνίας (MQTT, RabbitMQ), με τις σύνθετες δυνατότητες του Apache Kafka και του οικοσυστήματος του.

Σκοπός μας είναι η καταγραφή της ποιότητας της ατμόσφαιρας και η διανομή αυτών των δεδομένων σε οποιοδήποτε από τα υποσυστήματα της πλατφόρμας «SYNAISTHISI». Η προσθήκη του Apache Kafka και των παρεμφερών εφαρμογών του μας δίνει τη δυνατότητα να φιλοξενήσουμε στην πλατφόρμα ακόμα πιο συνθέτες cloud εφαρμογές. Συνεπώς αναπτύσσοντας την εφαρμογή για την ατμοσφαιρική καταγραφή, έχουμε τη δυνατότητα να αναλύσουμε σε βάθος τις τεχνολογίες που αξιοποιήσαμε κατά τη υλοποίηση της. Αυτές οι τεχνολογίες μπορούν να χρησιμοποιηθούν και σε άλλες εφαρμογές, ακολουθώντας το βασικό όραμα της πλατφόρμας «SYNAISTHISI».

Λέξεις-κλειδιά:

Διαδίκτυο των Πραγμάτων, $PM_{2.5}$ σωματίδια, SYNAISTHISI, cloud, ατμοσφαιρική ρύπανση, Apache Kafka

Table of contents

List of figures

List of tables

Abbreviations

IoT	Internet of Things
DevOps	Software Development and IT Operations
D2C	Device-to-Cloud
D2D	Device-to-Device
KRaft	Kafka Raft metadata mode
AMQP	Advanced Message Queuing Protocol
MQTT	Message Queuing Telemetry Transport
TCP	Transmission Control Protocol

Chapter 1

Introduction

In recent years, we've seen a significant rise in the use of the internet and technology applications in our daily routines. As we increasingly rely on smartphones and other devices, they are making our lives more convenient, adaptable and in some cases, even safer. A vast array of applications and services are readily available to us, ranging from simple payment and ticket booking systems to high-accuracy GPS maps and numerous other innovative technologies[1]. These technologies can be developed by both leading tech companies and smaller-scale initiatives. Often, these smaller companies or groups of developers aim to implement the well-established and ever-expanding applications of the Internet of Things network. This large network of connected devices is full of potential used to create new solutions that can meet many different needs. From creating smart environments that respond to user behavior[2, 3], to developing systems that monitor and manage resources for efficiency, such technological innovations are pushing the boundaries of what's possible with IoT technology.

The Internet of Things, commonly referred to as IoT, involves the interconnection of a vast array of devices and services via the internet, with the ultimate aim to orchestrate collective operation.[1, 4]. This interconnection is designed to generate applications and internet solutions for a wide range of scenarios. Such applications can be found in smart homes, healthcare, agriculture, transportation, energy management, various types of automation and of course, environmental monitoring,[5, 6, 7] which is the focus of our thesis. The broad scope of these implementations aims to enhance the quality of our lives, whether we're discussing large-scale applications or smaller companies seeking straightforward solutions that require internet interconnectivity. The rapid expansion of such applications, coupled with the vast

open-source community, can lead to the development of solutions for a wide array of challenges. This is particularly beneficial in certain areas that require more expedient solutions to address their unique local problems.

Given this local problem-solving approach and the accessibility of smart internet tools provided by IoT, we see the potential for creating widely available applications focused on environmental monitoring and air quality. Air quality and environmental pollution have been major concerns for modern societies and political spectrum in recent years[8]. The industrial revolution of the past century, along with the continued use of methods with high carbon and particulate matter footprints, have raised significant concerns, especially in recent years[9]. From a governmental perspective, particularly in Europe [10], there has been significant effort to revolutionize industrial production methods to be more efficient and environmentally friendly. Despite these valuable and substantial efforts to reduce air pollution, the problem is far from being fully resolved. These circumstances lead the way for innovative ideas to monitor air quality and pollution in efficient and affordable ways, primarily making them widely available to people concerned about such issues. This is where the extensive capabilities of IoT applications come into play, providing the opportunity to effectively monitor the overall air pollution footprint at a specific location[11].

1.1 Thesis' Vision

In our approach, we aim to create a highly interoperable and scalable solution that follows to the core principles of IoT. This is achieved by enhancing the capabilities of the SYNAISTHISI platform with emerging technologies and network protocols increasingly adopted across a broad range of technological infrastructures. SYNAISTHISI platform [12] initiated the contemporary concept of interconnecting diverse resources for various purposes, complying with the fundamental framework of IoT applications. Its basic idea was the creation of an IoT middle-ware that links a vast array of services, aiming to accomplish any given task: A service hosting platform where various system applications can be developed and operated by users in an easy and scalable manner. Also, the support of multiple protocols across different platforms, systems and services, makes the process of interconnecting elements like sensors, actuators and embedded systems more accessible. This combination of older protocols with newer ones enhances the scalability of our application and facilitates

its accessibility for future upgrades and covers a wide spectrum of use cases.

In this regard, we try to bridge our concern and the fundamental right of everyone to access information about the air quality in their living environment. More specifically, there has been significant interest in $PM_{2.5}$ particles in the atmosphere, along with general factors like temperature and humidity and their long-term effects on human health[13, 14]. We aim to provide an easy-to-use service for everyone, but most importantly, to encourage technologically savvy individuals to explore the power of existing IoT and open-source infrastructures. With some insight, they can create their own services and conduct their own research on any given problem. This luxury of leveraging existing Cloud infrastructures[15] and adding services to them that can address a wide range of problems is relatively new. Our approach lays the groundwork for innovative methods of developing applications that effectively enhance our lives and keep us informed about existing situations, all while adhering to the principles of the IoT philosophy.

This thesis is embarking on the task of implementing enhancements to an already established platform, which is recognized for its focus on the interoperability features of IoT ecosystems. The enhancements involve the integration of Apache Kafka[16], a continuously evolving technology, along with its associated technologies. Our objective is to expand and upgrade the SYNAISTHISI platform, by incorporating an advanced system that propels the process of data transfer and flow to the next level. Apache Kafka, with its event streaming capabilities across various systems, leads the way for fast and responsive IoT applications. Its high throughput, high availability and low latency make it an ideal choice (cf. Sec. 3.2). With the proper configuration and infrastructure, Kafka can transform small-scale applications into large-scale projects. By incorporating new components, primarily from the Apache project, we can seamlessly interconnect existing protocols and devices with newer ones in an easy and accessible manner. We persist in the fundamental approach of upgradability and expandability, thus ensuring the creation of an IoT platform capable of hosting multiple services and ensuring compatibility with future services across a broad range of use cases.

1.2 Contributions

The contributions of this thesis are summarized as follows:

1. Design and creation of a comprehensive cloud-based IoT Air Quality monitoring application.
2. Enhanced the SYNAISTHISI platform's event-driven capabilities through the integration of Apache Kafka, Kafka Connect and Schema Registry.
3. Implemented Kafka Connect to interconnect events across multiple protocols, and employed Schema Registry for efficient data validation and message optimization.
4. Develop and configure a high-performance, fault-tolerant and scalable Apache Kafka cluster.
5. Implemented a user-friendly and contemporary web application for real-time air quality data consumption on a live map, utilizing Web-Sockets and React.

1.3 Thesis Organization

The subsequent sections of this thesis are organized as follows: Background, motivation and related work are provided in Chapter 2 (cf. Chap.2), setting the theoretical approach. A detailed analysis of the utilized frameworks and services is presented in Chapter 3 (cf. Chap.3). A comprehensive description of the application's architecture, development process and implementation, including the design of our front-end map, is offered in Chapter 4 (cf. Chap.4). Experimental evaluation and bench-marking of the application, providing an in-depth performance analysis, are the focus of Chapter 5 (cf. Chap.5). Finally, potential future work and expansions, which concludes the thesis with reflections and directions for further research are discussed in Chapter 6 (cf. Chap.6).

Chapter 2

Background and Motivation

2.1 Environmental Perspective

In recent decades, there has been significant concern about various forms of pollution on our planet. Air pollution, in particular, has been a major issue for many years, especially in the most industrially advanced countries. A significant consequence of air pollution is the increase in particulate matter in the atmosphere, also known as $PM_{2.5}$. These particles can lead to various health problems, including cardiovascular, respiratory and neurodegenerative diseases [14]. This, in conjunction with the increasing global temperatures [13], has shown that even a slight increase in particulate matter pollution can have significant effects. Numerous advanced countries and regions are making strides towards reducing air pollution by not only mitigating [10] the factors causing this pollution, but also by enhancing the techniques for monitoring and analyzing air quality.

A prime example of efforts to address this situation is South Korea [17] which is using advanced monitoring and notification technologies to alert its citizens when high levels of air pollution are detected. This includes the involvement of technology companies, like IQAir [17], which provides high-performance real-time air quality monitoring systems to help combat the country's serious air pollution issues. When air pollution levels surge, citizens are alerted to limit outdoor activities and are advised to use masks capable of filtering particulate matter as a protective measure. Moreover, the initiative to adopt more advanced technologies for tracking the sources and impacts of air pollution has already started to show results, as evidenced by the reduction in the annual $PM_{2.5}$ concentration level [18], in the past years starting from 2019 to 2022.

2.2 IoT Perspective

Since such efforts require substantial government and enterprise funds, there has been a push for more accessible, simplified and cost-effective solutions. Internet of Things (IoT), coupled with the widespread impact of the internet on our daily lives [1, 19] over the past decade, has emerged as a central instrument for these accessible solutions. The IoT principles has evolved over the years, finding its place in numerous implementations and applications [19, 7]. It utilizes a broad spectrum of device-to-device and device-to-Cloud implementations [20]. Device-to-Device (D2D) communication, devices directly exchange data, optimizing local processing efficiency, but limitations arise in terms of range, scalability challenges and potential security vulnerabilities. Device-to-Cloud (D2C) implementations leverage cloud resources for scalability and cost-efficiency, offering centralized management, but may introduce latency, security concerns and dependence on reliable internet connectivity [21]. However, challenges in Device-to-Cloud interactions require careful selection and configuration of the frameworks for a robust implementation. This has paved the way for innovative business models and applications, promoting both technological advancement and economic growth in cities and regions that have embraced this approach in recent years [4, 22].

A bright example in our local environment in the city of Volos, Greece is GreenyourAir [23]. This IoT-based application employs simple and cost-effective sensors to measure the $PM_{2.5}$ concentration level, along with the temperature and humidity, initially in the city of Volos and more recently in other areas as well. It is, of course, utilizing Cloud infrastructure to store and process the real time data coming from affordable sensors. The use of smaller, cost-effective sensors, instead of large stations, facilitates the expansion of the existing network in a cost-efficient way. This approach broadens the availability and accessibility of air quality information to the public. In our thesis, we utilize a portion of their measurements as our data ingestion source, providing real-world data for our application.

2.3 Open-source Approach

To proceed further, it's essential to understand the significance of open source in general , as the entire expansion of our SYNAISTHISI platform is based on the implementation of open-source tools. Open source projects possess huge influence in the development process of IoT, web applications and projects in general, as they raise a collaborative environment where developers contribute their expertise, leading to robust and innovative solutions [24]. This continuous integration accelerates the development process, as issues are identified and resolved swiftly. Furthermore, open source projects provide a lot of learning opportunities for developers through a wide range of levels, while also ensuring transparency, as the source code is openly available for review, improvement and modification. Additionally, transparency enhances trust and allows for the customization of applications to meet specific needs, many large companies [25] from various industries to contribute to open source projects, aiming to develop frameworks and services in a more efficient and cost-effective manner. In the IoT field, open source projects have been key in pushing progress, providing many libraries, frameworks and tools [26] that simplify the development process, which combined with the ongoing support for these tools, enhances the scalability and future-readiness. In conclusion, most of the technologies we utilize for our implementation are originated from well-know open-source projects, that provide advanced development tools and frameworks, enabling the appreciation to the significance and value of open source for both small and large scale applications.

2.4 SYNAISTHISI Platform

The primary platform that we are enhancing in this thesis is SYNAISTHISI [12]. It offers the interoperability, scalability and interconnectivity that our application requires. There is a wide variety of supported protocols in the platform, but in this instance, we are primarily focusing on interconnecting Kafka and its associated tools with MQTT and AMQP protocols, through the integrated RabbitMQ brokers that are already supported by the platform. This strategy aligns with the principles outlined in the open-source approach section (cf. Sec. 2.3), as we aim to expand our application and integrate it with services that continue to receive support from their development communities, making our project more open and future-proof. In addition, the containerized nature of the services within the SYNAISTHISI platform

facilitates an easy way to upgrade the existing infrastructure with Kafka tools and enables a smooth transition to a production environment. Thus, we preserve the IoT perspective of our application and expand it with advanced yet open-source services.

2.5 Kafka Integration Large Scale Event Streaming

Real-time event streaming and processing, involves transmitting events and messages in a distributed system, allowing for reliable, low-latency and high-throughput data transfer [27]. Apache Kafka [16], is a widely used open-source stream-processing software, designed for large-scale data collection, processing, storage and analysis [28]. Since its launch in 2011 [16], the top three tech companies - Amazon, Microsoft and Google - have developed their own large-scale event streaming services: Amazon Kinesis [29], Azure Event Hubs [30] and Google Pub/Sub [31], respectively, inspired by the Apache Kafka project. We note that these are paid services on fully managed Cloud provider platforms [32]. However, by utilizing Kafka, we gain access to many features typically found in such paid services, along with the large-scale, high-volume and fault-tolerant data ingestion solutions that Kafka offers. Therefore, with the proper configuration and understanding of our application's needs, we can still leverage these capabilities using this widely adopted and contributor-maintained open-source service.

Chapter 3

Available Frameworks and Related Work

There have already been made significant studies around the subject pf IoT platforms in general and, more specifically, IoT-based solutions for air quality applications too. Projects like SEMAR [33], which is similar to the SYNAISTHISI platform, aim to develop and maintain their implementations based on IoT principles. Additionally, they provide an interface capable of hosting multiple IoT applications. Papers, such as the one in [34], aim to extend the SEMAR platform with similar implementations, focusing on developing an air pollution monitoring application that leverages real-time air quality detection and integrates tools from the Kafka ecosystem. Similarly, in paper [35], there is an effort to develop a similar application using various tools developed by Apache and other infrastructures focused on IoT. Our objective differentiates us from these efforts, by aiming to develop a platform that not only accommodates to our air quality requirements by supplying all essential components for an air quality application, but also ensures our platform remains scalable, upgradable and highly available across different hosting environments. We leverage technologies, services, methodologies and configurations originating from the enterprise development business (e.g. data validation, fault tolerance, high throughput) and incorporate them into our IoT and open-source based application.

Progressing, another instance is the development of an air pollution monitoring system with sensors on city buses [36]. Another one MoreAir [37] presents an agile and low-cost urban air pollution monitoring system. These provide some comprehensive solutions and implementations of the innovative idea of installing sensors on city buses to gain a more detailed and focusing on low and accessible cost of such infrastructures. While acknowledging the potential of such efforts, some studies analyze the challenges that they have faced [38],

such as the lack of universal applicability of these implementations for specific purposes. Therefore, a significant part of our focus is on the creation and expansion of an upgradable and expandable platform, able to host IoT based applications similar to air quality monitoring. In this regard, we have avoided developing an overly specialized implementation exclusively for our air quality application. Apart from merely implementing and completing the monitoring task, we also emphasize on the services and tools we are integrating into our platform. We develop and optimize our infrastructure for environmental monitoring, showcasing the re-usability of individual services of our stack across similar infrastructures. We aim to categorize our stack in a way that allows groups and layers of our system to be efficiently maintained and upgraded (cf. Sec. 4.1). Lastly, compared to [34] and [35], we place a high priority on implementing and developing services maintained by their contributors and the open-source community. This approach ensures our platform remains future-proof and up-to-date with emerging technologies.

3.1 Docker

The primary development environment utilized in this thesis is Docker [39]. All applications are executed in a containerized environment using the Docker engine, a tool renowned in both the IoT and enterprise markets for its efficiency and reliability. Docker is a platform as a service product that provides the ability for services to run on an operating system level virtualization, as shown in Figure 3.1. Unlike traditional virtual machines, containerized services such as Docker provide enhanced resource efficiency and uniformity across various environments, making them an optimal choice for modern software development due to their reduced system demands and consistent performance.

Alternative solutions to Docker are Podman, Containerd and LXC (Linux Containers), all of which offer container runtime services. However, Docker stands out due to its simplicity and large user community, that makes it a more accessible and user-friendly option for developers. By adopting this approach, each individual service is isolated from the others , creating a conducive development environment. This allows developers to focus on their workflow for individual services without the concern of compatibility issues, thereby enhancing productivity and efficiency.

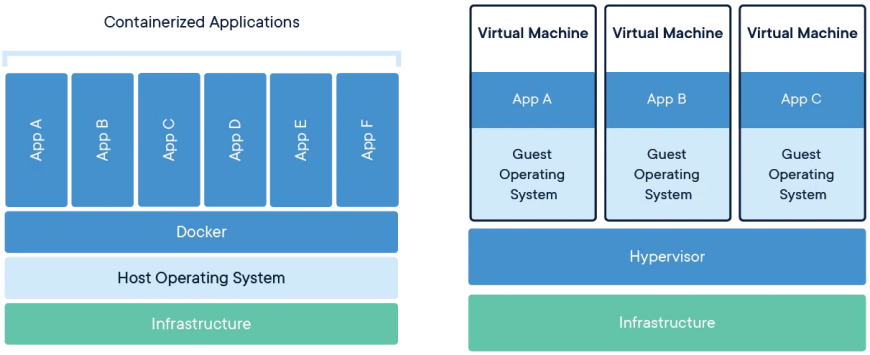

Figure 3.1: Comparing Containers and Virtual Machines. [Ref]

3.1.1 Docker network

Containers, as they are referred to, represent these isolated environments. They are interconnected through the advanced Docker network [40], which offers a wide array of options and configurations, ensuring high availability for any given setup. This isolation, while also allowing communication between the containers when needed , facilitates the creation of highly scalable and secure applications for future upgrades and optimizations. In contrast to traditional virtualization, Docker images and their associated containers can be incorporated into other applications with minimal configurations, making the project easier to deploy and even adjust in environments that are already deployed.

3.1.2 Docker compose

Docker Compose [41] is a tool that enables the operation of multi-container Docker applications. It serves as the connecting link, providing a simple and organized interface for running multiple containers in conjunction with each other. It offers an ideal environment for both development and smaller scale production applications, allowing developers to have an organized view of their application's configuration, networking and simplifying the development process. Through its interface, developers can coordinate both official and custom images, integrating them into their application as needed. Docker Compose aligns with the DevOps principles, aiming to bridge the gap between the development and the actual functionality of an application.

Today's leading cloud service providers offer near-native support for containerized applications, making Docker tools an excellent choice for managing the development process. This support significantly simplifies the transition from simple IoT implementations

to production-level deployments on a cloud provider, making the experience more seamless and intuitive.

3.2 Event Streaming

Event streaming is the process of capturing real time data from a variety of event sources like sensors, devices, gateways, databases and other cloud services. Apache Kafka [16] is a relatively recent technology that introduces the concept of event streaming to the cloud community. The main flow scheme utilized here is publish/produce, store, subscribe/consume. This model is versatile and can be applied to nearly any type of cloud communication , its primary function being to handle large amount messages per second.

3.2.1 Apache Kafka

Apache Kafka provides low latency, high throughput, scalability and high availability. Kafka enables the use of high-volume real-time data transfer with low latency and provides temporary storage for short-term usage. In contrast, the conventional database model requires all produced data to be first stored in the database before it can be used for any given task. Kafka can be viewed as a cloud cache, offering high availability and low latency , which makes it an excellent choice for real-time event streaming applications. Event streaming is leveraged across a wide spectrum of use cases, including activity tracking for real-time insights, processing data in real-time for immediate decision making, providing low latency and fault-tolerant messaging for critical systems , monitoring operational data to maintain system health and aggregating logs for efficient analysis and troubleshooting.

3.2.2 Events

Events or event records are the messages that contain the information about an actual event that happened (in a structured manner) and they are the main way of reading and writing data to Kafka. An event mainly consists of a key, a value and a timestamp. Each event within the same topic is distinguished by a unique key. The key determines how these events are distributed across different partitions (cf. Sec. 3.2.4 3.2.5). The value is the actual data that is distributed and varies from simple text, to JSON file and even more complex formats like

Avro [42], that are encoded properly and stored in the Kafka topics. The timestamp contains information about the time that each event took place.

In the following example, the key 'MqttSensor1' uniquely identifies the source of the data produced or consumed by a Kafka topic. The value 'Mqtt sensor 1 measured $25°C$' represents the actual data recorded and the timestamp 'Jul. 15, 2022 at 12:56 a.m.' indicates when this measurement was taken.

- Key: 'sensor1'
- Value: 'Sensor 1 measured 22.5 °C'
- Timestamp: '2/3/2024, 3:14:23 PM'

3.2.3 Brokers

Brokers serve as the primary infrastructure where data is temporarily stored for subsequent processing, before being pushed for long-term storage. Brokers are organized into clusters, allowing for multi-broker and multi-cluster setups. We do not delve into multi-cluster implementations, since they are primarily used in large-scale multi-region enterprise applications [43]. The complexity and operational overhead of managing multiple clusters is not justifiable for most IoT applications, since our single-cluster implementation meets the requirements for high throughput, fault-tolerance and low latency.

In the past, Kafka was utilizing Zookeeper [44] to orchestrate the communication between the Kafka broker nodes and the metadata management. But now, a new protocol called KRaft(Kafka Raft metadata mode) [45] has been implemented to act as the new quorum controller and has a more direct connection to all the Kafka nodes. Nodes can function as a controller, a broker, or both, as depicted in Figure 3.2. Controller nodes , deployed as Kafka images, play a crucial role in storing all the necessary metadata for managing the distribution and synchronization of data across the broker nodes.

Applying the correct configurations, each Kafka cluster in a multi-broker implementation becomes fault tolerant. Kafka controllers are responsible for the load balancing and the fault tolerance of any Kafka application. Furthermore, Kafka's ability to provide high throughput and low latency ensures that applications using it are not only fast and responsive, but also safeguarded against data losses due to run time failures. Multi broker implementations combined with the new KRaft control protocol offer faster response times whenever data recovery

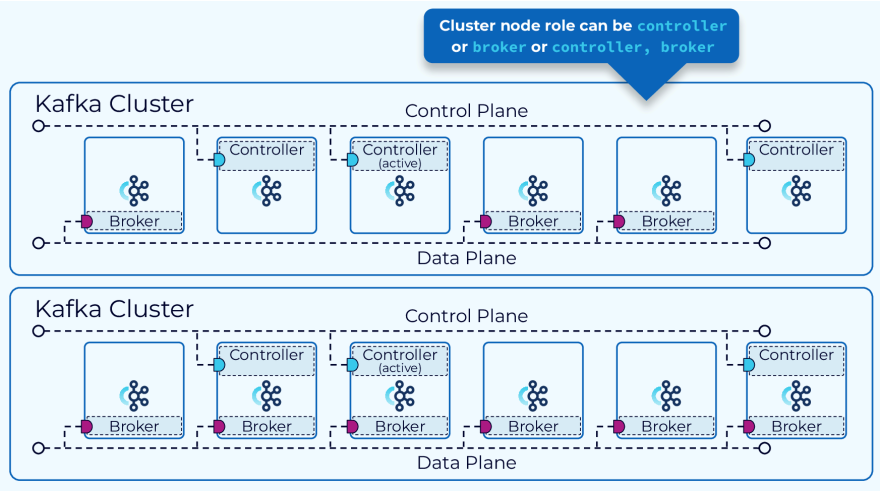

Figure 3.2: KRaft Cluster Node Roles. [Ref]

is needed, which usually occurs in the case that one or more brokers fail during their operation. In alignment with the Apache Kafka project's current recommendation, we have opted to use KRaft as the controller in our thesis, replacing Zookeeper, which has been deprecated for new deployments [46].

3.2.4 Kafka Topics

Going a step further, Kafka uses topics to manage and securely store events. Topics function like folders in a file system, with events acting as the files within. New event messages are appended to the end of the topic and, once written, these events are unchangeable. In Kafka topics, unlike other messaging systems, events are not deleted post-consumption. Instead, a retention period is set, beyond which messages are deleted, with the maximum duration being up to one week. The performance of Kafka remains stable regardless of data stored, implying that long-term data storage should have a minimal performance impact.

3.2.5 Kafka Partitions

Kafka topics are segmented into partitions. A single topic in Kafka is divided into multiple partitions . Partitions are distributed across Kafka brokers in the cluster and messages are shared among multiple nodes. The performance of reading, writing and processing events is maximized and data is replicated across all Kafka nodes, presented in Figure 3.3. New events are appended to a specific partition within a topic and events with the same key are stored in the same partition, ensuring data ordering. This partitioning mechanism contributes to the

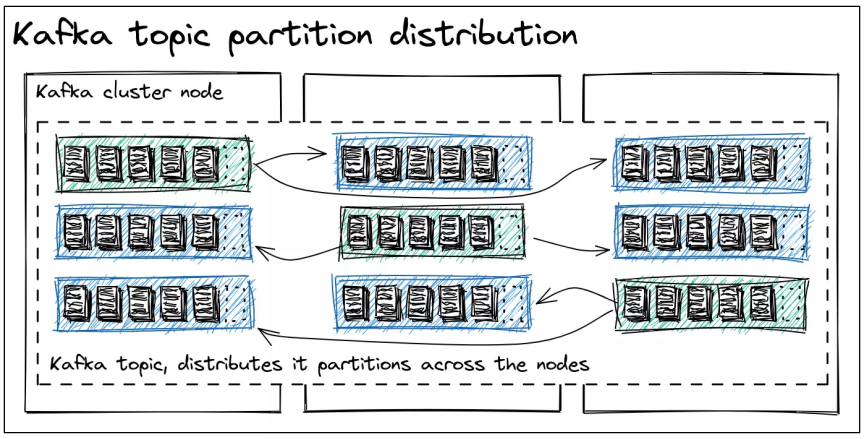

Figure 3.3: Kafka topic partition distribution. [Ref]

high throughput and fault tolerance capabilities of Kafka.

3.3 Existing Event Streaming Protocol Integration

We support different protocols currently available on the SYNAISTHISI platform. Since most IoT devices and applications primarily use the RabbitMQ and MQTT protocols, our primary focus is to interconnect these protocols with our Kafka components.

3.3.1 RabbitMQ

RabbitMQ [47] is an open-source message broker, also known as a message-oriented middle-ware. It implements the Advanced Message Queuing Protocol (AMQP) and supports several other protocols. RabbitMQ provides robust messaging for applications. It is easy to use, runs on all major operating systems and supports a huge number of developer platforms. It also provides a wide range of features to let you trade off performance with reliability, in means of persistence, delivery acknowledgments and high availability. The key supported protocols are AMQP, MQTT, STOPMP, HTTP and Web Sockets. Additionally, it offers multi-language client support, providing a wide range of development platforms of the most popular programming languages and frameworks like Python, Java, .NET, JavaScript, Ruby, Go, etc. RabbitMQ can be deployed in distributed and federated configurations to meet high scale and availability requirements. It offers flexible routing and bridging mechanisms, along with fault tolerance features such as clustering, automatic fail-over and message durability. This wide variety of protocol support and accessible bridging between those protocols make RabbitMQ

a valuable tool for the IoT market, while worldwide large-scale companies also use it, mainly for their messaging infrastructure.

3.3.2 MQTT

MQTT [48] stands for Message Queuing Telemetry Transport. It is a lightweight publish-subscribe network protocol capable of transporting messages. It is designed for restricted devices and low-bandwidth, high-latency, or unreliable networks. The protocol ensures reliable message delivery and minimizes network bandwidth requirements. It provides one-to-many message distribution applications and message retainment, since an MQTT topic holds the last messages and the next time a client subscribes, it immediately receives these messages. The low transport overhead and quality of service options provided by MQTT make it an ideal choice for scenarios where network bandwidth is limited and connectivity is unreliable. Thus, its lightweight nature and efficient use of network resources make MQTT a popular choice in the IoT ecosystem, where devices often operate under unstable network conditions and power limitations.

3.4 Kafka Connect

Kafka Connect [49] is a robust framework that allow us to interconnect our Kafka cluster with external systems such as databases, search indexes and different kinds of communication protocols. It is designed to handle various event streaming use-cases, including scalable and reliable streaming of data to and from Kafka. The framework is designed around a simple concept of connectors and workers. Source connectors import data from systems and propagate it to Kafka, while Sink connectors export data from Kafka to external systems. Connectors manage the integration between Kafka and other systems and they can be created without the necessity of additional code to be written. Each time we add a new external framework to our stack , we usually need to write the code that interconnects it with our Kafka cluster. However, this type of code does not provide any advantage to our application compared to other similar applications, especially in systems where a lot of services that need to communicate with each other are involved. This resolves much trouble during the development time of a specific application, as it removes the necessity of writing undifferentiated code for those external interfaces every time a new one is incorporated. All undifferentiated

code interfacing with other systems is bundled into separate libraries, allowing us to deploy only the necessary connectors. Furthermore, Kafka Connect is a highly acceptable solution for interconnecting different protocols to our Kafka implementation.

Connect uses numerous connectors, among which we utilize the RabbitMQ and MQTT source-sink connectors. This allows us to interconnect these protocols that already exist in the SYNAISTHISI platform with Kafka. If necessary, we could indeed create our own connectors, providing us with the flexibility to manage virtually any data integration requirement. Kafka Connect includes a REST API for managing and monitoring connectors through workers. This API allows us to deploy, remove, pause, resume and check the status of workers associated with their corresponding connector. Automatic offset management is provided, therefore in case a connector fails, the replacement can resume exactly where the previous one stopped, guaranteeing data consistency. Kafka Connect is a powerful tool that simplifies the process of integrating different systems with Kafka and empowers robust, scalable and reliable data pipelines in our implementation.

3.5 Schema Registry

Kafka Schema Registry [50] offers a REST interface for storing and retrieving schema registries and is an essential part of the Apache Kafka ecosystem. Its role is crucial in maintaining the consistency and well-defined nature of the data in Kafka topics. Kafka Schema Registry is used in combination with Apache Avro [42], providing a data serialization system that encodes data in a compact binary format . Avro schemas define the structure of the data being encoded, making it easy for each consumer to correctly deserialize the data. This advanced and compact binary format allows us to significantly reduce the event's message size, especially in instances where the producer is utilizing the JSON format.

The primary purpose of a schema registry, is storing and retrieving schemas in a centralized location, presented in Figure 3.4. Producers and consumers that utilize schema registry, own a contract of what type of data to expect, reducing the possibility of data inconsistency and parsing errors . Schema evolution is supported to accommodate the inevitable changes in our data over time, reflecting the evolving needs of our applications. This ensures our system remains flexible and adaptable to meet these ongoing changes. The support for schema versioning in our system allows us to maintain compatibility with older data that was encoded

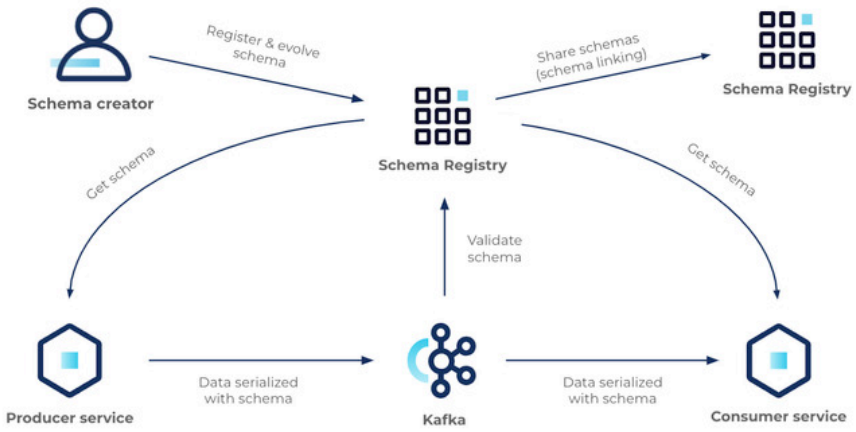

Figure 3.4: Schema Registry architecture overview [Ref]

using previous versions of the schema, providing seamless access and utilization of older data structures.

In summary, Schema Registry is a crucial tool for maintaining compatibility and data integrity in Kafka-based systems. It provides a centralized store for schemas, supports schema versioning and evolution and allows for a clear contract between producers and consumers. Over the past few years, it has emerged as an industry standard for event streaming applications, driven by the increasing need for data size reduction and compatibility in the face of growing data volumes and complex processing requirements.

Chapter 4

Architecture and Implementation

In this chapter, we are going to analyze and present the technical details of our project. We explore the critical aspects of our project, focusing on the implementation and execution of our designed solution. In addition, we provide details on the system's architecture, the chosen configurations and a further explanation of the key fundamentals that were prioritized for our implementation. Furthermore, we outline the main challenges encountered during the implementation phase and how we addressed them.

4.1 System Overview

Our system is divided into three primary layers: the data ingestion processes, the events streaming and storage and data consumption and distribution in our front-end map. The system's architecture is presented extensively in Figure 4.1.

Starting from the top, three protocols for producers are supported: native Kafka, AMQP and MQTT. A RabbitMQ broker within the SYNAISTHISI platform manages messages generated by MQTT and AMQP clients. Subsequently, Kafka Connect interlinks our events from the RabbitMQ broker with our Kafka Cluster. We primarily utilize source connectors for our air quality application, meeting our data flow requirements. The Schema Registry is mainly used for our native Kafka producers, in order to validate our data and minimize the message size sent to our Kafka cluster. Avro is utilized as the serialization format in our Schema Registry service.

Events are subsequently streamed into our Kafka cluster. Our cluster operates in KRaft mode and consists of four nodes. A single controller node maintains the health of our Kafka

cluster and performs tasks related to load-balancing and data recovery, in the cases that run-time failures occur. The remaining three nodes are data brokers configured in prioritizing data reliability, availability and ordering. In order to monitor and manage our Kafka Cluster and its associated services, such as Kafka Connect and Schema Registry, we have integrated an advanced control panel. AKHQ [51] provides us with a user friendly graphical interface, allowing us to maximize our operational capabilities in a modern and accessible manner.

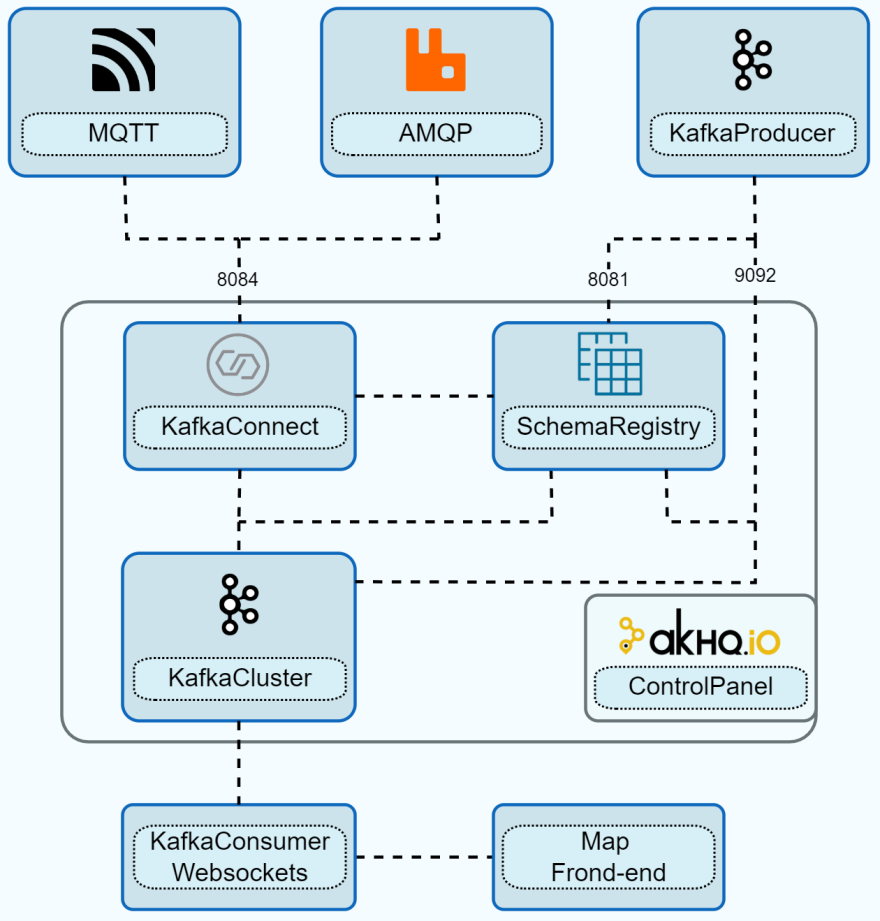

Figure 4.1: System Architecture

Additionally, a web application has been developed to display measurements on a live map. An automated process parses through the Kafka topics and assigns consumers to them. Both simple and Avro consumers are orchestrated to work in sync to fetch live events from our cluster. Web-Sockets, over a TCP connection, is continuously active application, ready to transfer data from our server side to our web browser. For our front-end we utilize React [52] and OpenStreetMap [53] to present the measurements on a contemporary, visually appealing map-based web page. Each point on the map represents sensors that continuously update and refresh their values. This configuration guarantees real-time and precise data representation,

while ensuring low latency, high throughput and optimal availability.

4.2 Data Ingestion

Continuing with the principles of the SYNAISTHISI platform, we provide support for the primary messaging protocols that are found in the majority of IoT infrastructures. This support allows us to establish connections with a wide variety of sensors, gateways and devices. By doing so, we enable the interoperability, scalability and interconnectivity principles that our application requires. Aside from our focus on air pollution measurements, our platform is capable of utilizing any type of data that is supported by these protocols. This flexibility helps us to adapt and configure our application for different kinds of implementations. Furthermore, we are not only confined to the typical messaging protocols that are commonly used in IoT applications, but utilizing more advanced protocols like Avro. Avro allows us to handle more complex data structures and provides us with additional capabilities, enhancing the overall functionality, versatility and upgradability of our platform.

4.2.1 Producers

We have created a set of producers for each protocol, aiming to simulate the respective clients that would publish our measurements to our brokers. These producers are written in Python in our implementation, but the operators of our application can choose any language pack depending on the compatibility of their devices and gateways. Our primary messages are based on the JSON file format, that provide us a broad system support and the ability to configure the rest of our infrastructure based on an internet standard message type. The majority of messaging and event-driven applications primarily use JSON due to its simplicity and widespread support across virtually all cloud infrastructures.

Additionally, messages are structured in JSON files containing the following fields: DeviceID, Latitude, Longitude, $PM_{2.5}$ concentration in micro grams per cubic meter of air ($\mu g/m^3$), Humidity in percentage (%), Temperature in Celsius ($^{\circ}C$) and Timestamp, as can be seen in Figure 4.2. DeviceID is the name or label of a given device that fetches measurements. Latitude and longitude represent the coordinates where each device is located and enable us to locate our device on our live map. If our devices support GPS, the coordinates should be imported to these fields, otherwise operators should input the approximate location. PM2.5,

temperature and humidity are the primary data needed for our air quality application. The Timestamp field provides us with the time and date when our event took place.

4.2.2 Interoperability Kafka Connect

The data produced by MQTT or AMQP clients, are directed to our SYNAISTHISI RabbitMQ server. In this manner, we have implemented Kafka Connect to fetch our data from our RabbitMQ broker to our Kafka cluster. We have integrated both source and sink connectors for MQTT and AMQP, therefore supporting bidirectional communication between RabbitMQ and Kafka. In our application, we utilize the source connector (which inserts data from an external system into our Kafka Cluster) for both MQTT and AMQP, as it aligns with our application's data flow requirements. This setup enables our Kafka infrastructure to interact with external systems in a straightforward and accessible manner.

Connectors are configured in distributed mode, which allows us to replicate the deployed workers, activating the fault-tolerant capabilities of Kafka Connect workers. It is worth noting that we deploy a single worker on each connector, not multiple ones, but users of our platform have the option to replicate the workers in case they want to protect the Connect workers from runtime failures. Our workers can be both schema enable or not, enabling the data validation routine for non Kafka producers as well. As discussed in Kafka Connect section (cf. Sec. 3.4), Kafka Connect allows us to interconnect our Kafka cluster with a wide variety of external systems and services. This eliminates the need to write additional code each time we integrate a new external service, thus significantly reducing our development time.

4.2.3 Data Validation Schema Registry

We also utilize the Schema Registry in our application, both for native Kafka producers and for Connect workers. Schema Registry allows us to implement a more compact communication protocol. The initial JSON messages are serialized into an efficient binary format, ready to be broadcast to the Kafka Cluster. This process enables us to reduce the size of the JSON events that are sent to our Kafka Cluster, decreasing the message size by 40% or more, depending on the size of the JSON files and the number of fields. The more fields a JSON file has, the greater the reduction in message size, thanks to the efficient compression in Avro. It requires the initial message to be in JSON format before being converted, ensuring a consistent event data structure whether we utilize the Schema Registry or not. One

advantage of Avro over similar protocols is its wide support across various communication and messaging infrastructures. This makes our platform highly efficient, sophisticated and highly configurable, depending on our implementation

In our implementation, our Kafka producers utilize Avro. This gives us comprehensive control over our data structure and the fields necessitated in our JSON files. We generate an AVSC file, which is a JSON format file that defines the schema or structure of the data to be serialized using Avro, as shown in Figure 4.2. For instance, if we desired to make the 'Humidity' field optional and the '$PM_{2.5}$' required, the message streaming would persist even if the former missing, but would stop if the latter was missing. In our implementation all fields are required, so we designate them all as mandatory. This provides a centralized way to supervise and monitor our data structure and compatibility in an organized and secure manner. Last but not least, data sent from our MQTT and AMQP connectors in our air quality application do not utilize the Schema Registry, as we want to simulate situations where we interconnect our platform to pre-existing sensors and aim to avoid developing new code when connecting them.

```
"deviceId": "sensor2",
"latitude": 39.36103,
"longitude": 22.94248,
"pm25": 12.5,
"temperature": 22.5,
"humidity": 60.0,
"timestamp": 164099520
```

```
"type": "record",
"name": "SensorData",
"fields": [
{"name": "deviceId", "type": "string"},
{"name": "latitude", "type": "double"},
{"name": "longitude", "type": "double"},
{"name": "pm25", "type": "float"},
{"name": "temperature", "type": "float"},
{"name": "humidity", "type": "float"},
{"name": "timestamp", "type": "long"}]
```

Figure 4.2: Presentation of our JSON file (on the left) and our Avro Schema (AVSC, on the right).

4.2.4 Data Structure

In the Events and Producers sections (see Sec. 3.2.2, 4.2.1), we discuss how our JSON messages are structured in both a general event message example and in our specific implementation. However, the format presented is the human-readable one, which is how the data will appear on our front-end page. The actual JSON files, when ingested into our Kafka cluster, will appear as shown in Figure 4.2. Most of the fields remain understandable and

human-readable, even in this form. The exception is the timestamp, as we utilize Unix epoch time, which represents a specific time as a single number. The Unix timestamp is widely adopted in most computing environments due to its compact size and simplicity for data storage and transmission. Therefore, we only need to convert it into a human-readable form when we intend to display it in our application, while our data structure remains highly scalable and flexible.

4.3 Kafka Cluster Configuration and Management

During the development of our Kafka Cluster, we faced a series of challenges. Our primary objective was to create a platform that would be highly available and easily upgradable for integration with other applications too. The configuration we chose for our Cluster reflects that of a production environment, minimizing the need for operators to make further extensive configurations when hosting their own applications on this part of our platform. We designed an implementation that aligns closely with enterprise-level features and requirements, while maintaining IoT principles of simplicity and ease of use.

Our implementation is developed to seamlessly interconnect with our existing SYNAISTHISI platform , ensuring a unified and interconnected system. We leveraged the latest technologies to optimize our Cluster's performance, achieving high-speed data streaming and processing without compromising its robust fault tolerance capabilities. This balance between speed and reliability is a critical aspect of our project, as it ensures our platform can handle large volumes of data while maintaining durability during runtime failures. By addressing these challenges, we have created a Kafka Cluster that is not only powerful, scalable and reliable but also adaptable and user-friendly, making it a valuable tool for anyone who wishes to take advantage of it.

4.3.1 Cluster Configuration

Our cluster is structured in a four-node Kafka implementation, as depicted in Figure 4.3. Our nodes function either as controllers or data brokers, with a single node and three nodes serving as the controller and data brokers respectively. This is the recommended production implementation by the Apache project, as combined controller/broker nodes are primarily used for presentations, early development and testing purposes. This setup allows us to

leverage the high throughput and fault tolerance features that Kafka is capable of. Furthermore, depending on the demand of our application, we have the flexibility to scale our cluster from three brokers to five, seven and beyond. This scalability ensures that we can meet the increasing demands, if needed, without compromising the performance or reliability of our system.

Instead of opting for the deprecated Zookeeper control plane, we employ the relatively new KRaft protocol. Adopting KRaft significantly increases the speed of data recovery and re-balance time when a broker node fails during operation and it provides better metadata management compared to Zookeeper. The internal configuration we have established allows us to expose only a single port (9092) of our cluster. The rest of the data flow is contained within our cluster, eliminating the need to expose our system further. This approach enhances the security of our implementation while maintaining its efficiency and effectiveness, as shown in Figure 4.1.

4.3.2 Control Plane

Furthermore, we utilize a single controller node in our control plane without replicating it, as seen in Figure 4.3. This approach helps us avoid unnecessary complexity and overhead, without direct exposure to the risk of data loss. We need to note that this decision was made after careful consideration of our requirements and priorities. While replicating the controller node could theoretically increase fault tolerance, it would also introduce complexities and potential synchronization issues. Instead, our focus on fault tolerance is achieved through replicating only the data brokers. This way, even if our controller node goes down during runtime, our data within the brokers remains safe. By leveraging a single node control plane configuration we achieve a firm balance between simplicity and robustness in our Kafka implementation.

4.3.3 Data Plane

Our three broker data plane endows our system with enhanced reliability, fault tolerance and scalability. Firstly, this configuration offers superior fault tolerance compared to single and two-broker clusters. In the event that one of our brokers fail, Kafka will redistribute the load across the remaining two brokers. Since we are utilizing KRaft, the time needed for our Kafka Cluster to re-balance the replicated data is significantly minimized. The residual

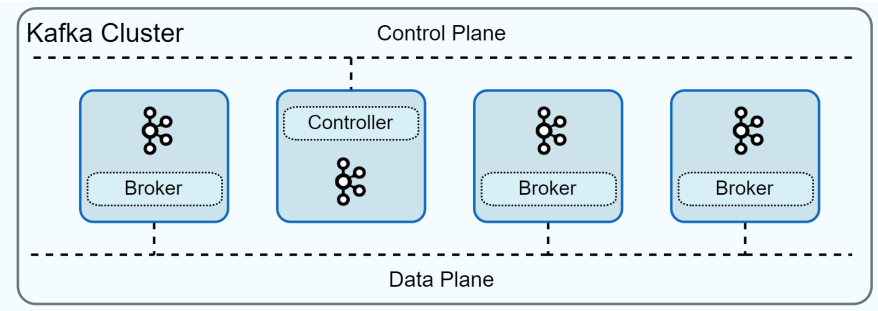

Figure 4.3: Cluster Configuration

two nodes are able maintain the data flow, ensuring minimal downtime during our cluster and application operation. This approach guarantees high availability for our platform, preventing disruption to our data streaming process in case of a broker failure.

Moreover, in terms of scalability, a three broker configuration is beneficial as the workload is distributed across multiple nodes. This arrangement distributes our data across each broker, ensuring a balanced workload. If the data volume and processing requirements escalate, we can increase the partition count on any given topic. Given the design of our data plane, we have the flexibility to easily expand our infrastructure with additional brokers if necessary. This ability to scale horizontally allows us to adjust to changing workload demands and ensures our platform is prepared for future expansion and enhancements. In this way, we have achieved an optimal balance between fault tolerance and scalability, making our implementation adaptable to a broad range of use cases.

4.3.4 Control Panel

Last but not least, we have implemented a control panel using AKHQ, which provides a contemporary and accessible graphical interface for managing our Kafka Cluster and its associated services. In addition to the SYNAISTHISI platform interface, operators can utilize this control panel to create topics, seen in Figure 4.4. It also facilitates schema creation, modification and evolution within our Schema Registry service. All Kafka Connect workers can be deployed and controlled from this panel, simplifying monitoring and management during application runtime. This eliminates the need for users to rely on CLI (Command-line interface) tools, offering a more modern management method. As both self-managed and fully-managed cloud service providers rely on graphical interface control panels for their operations, our approach aligns with this standard. AKHQ is based on an open-source project

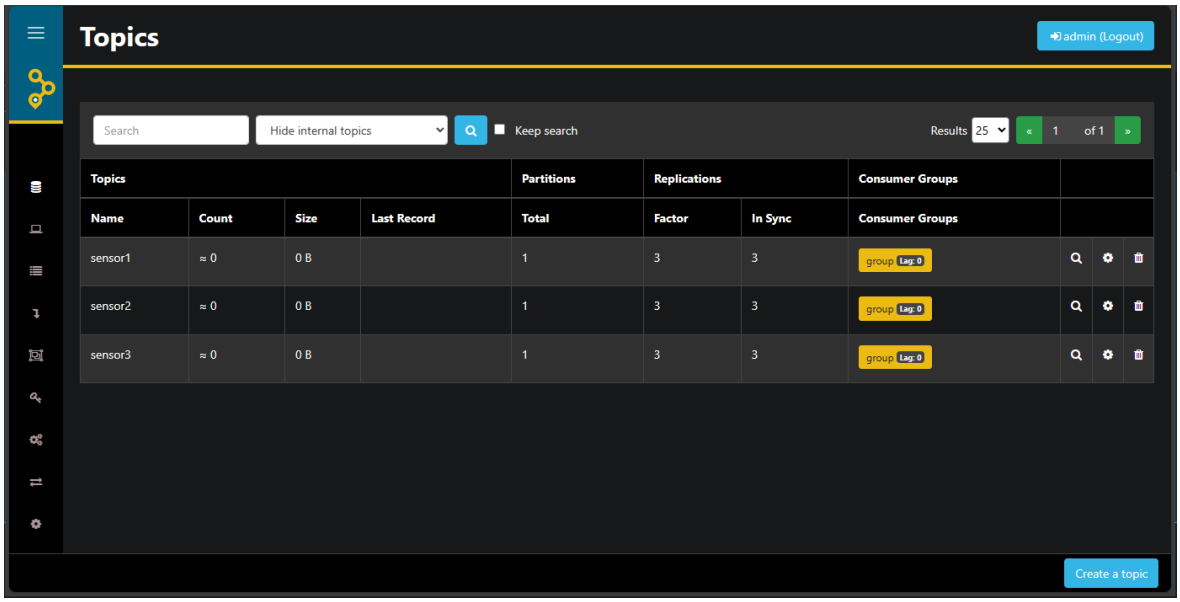

Figure 4.4: Control Panel Overview

with numerous contributors, guaranteeing its ongoing support and maintenance.

4.4 Web Application

In this section, we analyze the creation of a server-less side, back-end service and front-end page. These components fetch data from our cluster and display them on a live map. This implementation is specifically designed for our air quality application and is not a general-purpose platform like the Data Ingestion and server-side parts. On this regard, we have implemented a service that consumes live data from our Kafka Cluster and automatically detects the addition of new devices in our cluster. The consumed events are then sent over Web-Sockets to our front-end map, which is hosted as part of our modern-looking React live page.

4.4.1 Kafka Consumer

Whenever a topic is added to our server-side, the back-end service automatically detects the newly added topic. Each sensor, device, etc. on our map is uniquely identified by the key that has been assigned to it. Kafka consumers then read the incoming live events and list them in a batch of messages. In our application, the consumption rate is every five minutes, but this can be modified to any preferred frequency. Consumers can either be simple Kafka consumers or schema-enabled ones, as our universal JSON event message structure allows

us to utilize both without any incompatibilities. This highlights the importance of having a firm and well-thought-out structure for our event streaming data, as it allows us to consume and process our events efficiently, eliminating the time required for post-consumption data modifications from the cluster.

4.4.2 Web-sockets Integration

To facilitate the interconnection between our back-end service and front-end map, we utilize Web-Sockets. Web-Sockets is a protocol that unlike traditional HTTP connections, establishes a continuous communication channel over a single TCP connection between the front-end and back-end. Web-Sockets does not have any limit on the batch size that can be transmitted, making it an excellent choice for our needs of continuous data flow for the events consumed by our Kafka Consumer service. This ensures flawless real-time data transmission and enhances the responsiveness of our application. Furthermore, we minimize the need for frequent HTTP polling, which enhances our application's network efficiency. Therefore, by leveraging Web-Sockets, we can effectively align with the high-volume and real-time data streaming capabilities that our Kafka infrastructure is offers.

4.4.3 Front-end Map

Our live front-end map is developed utilizing React, a modern and efficient development environment. This framework allows us to create a visually appealing and highly functional web page, ideal for displaying real-time data. The map we utilize is based on Leaflet OpenStreetMap, an wide spread open-source map solution that perfectly suits our application needs. In addition to its efficiency, The component-based architecture of React enables us to easily manage and update our application's interface, ensuring a smooth user experience.

Our web page is divided into two primary sections, the map section and the data section. In the map section, Figure 4.5,each point represents a sensor at a specific location. When a point is selected, it displays the measurements from the corresponding sensor. As detailed in the Data Structure section (cf. Sec. 4.2.4), our measurements include the device name, the time of the measurement, $PM_{2.5}$ concentration ($\mu g/m^3$), Humidity (%), Temperature ($^{\circ}C$). The data is presented in a bracket when each point on the map is selected. Whenever a new device is incorporated into our platform, a new point is automatically created on the map according to

the device's coordinates. Consequently, this way our map page maintains a dynamic, user-friendly and modern design.

The other section of our page features a set of dashboards, Figure 4.6, each designed to display the three key measurements from our sensors: particle matter concentration, humidity percentage and temperature. Each set of tables within the dashboards corresponds to a specific point on the map. As these points are dynamically updated on the map to reflect the real-time locations of our sensors, the relevant tables in the dashboards adjust accordingly. This ensures that the set of data displayed is always updated. Moreover, the system allows us to display measurements from various time frames. Users can view measurements from a single day, three days, or a week old, providing them a comprehensive view of understanding of fluctuations over these periods over these periods.

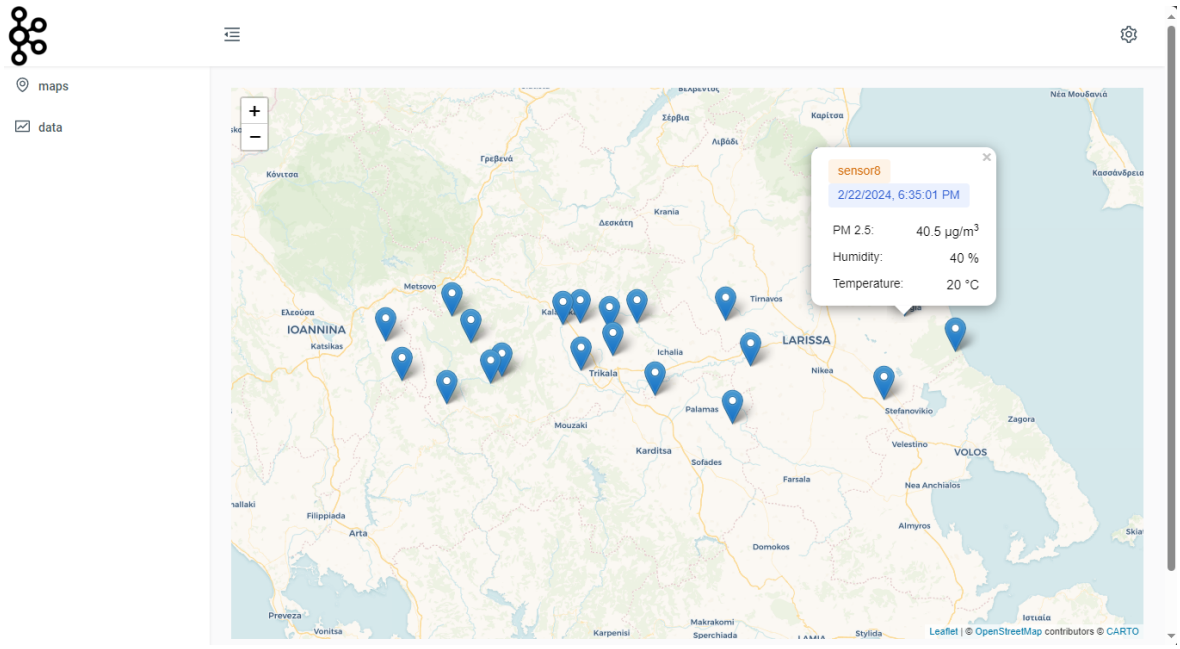

Figure 4.5: Live Map Overview

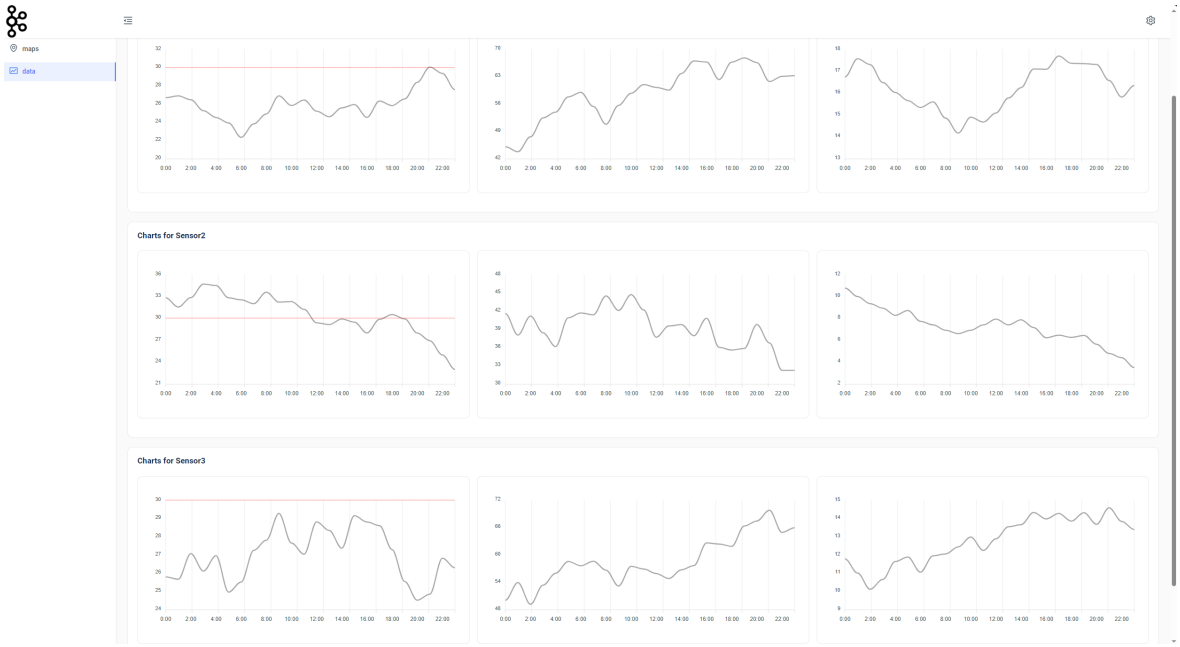

Figure 4.6: Historical Data Dashboard Overview

Chapter 5

Experimental Evaluation

At this point, we have described our primary motivation and background, and comprehensively analyzed the architecture and implementation of our application. In this section, we execute a series of realistic routines to demonstrate the key principles of our implementation. We test the data validation process and message size reduction. Finally, we deploy a realistic number of producer clients, generating data for our Kafka cluster, so that we can evaluate its throughput.

5.1 Data Validation Routine

In order to test our data validation routine we have prepared a set of scenarios, configuring our schema, Figure 4.2, depending on the use cases and our needs.

1. Firstly, we can configure all fields in our schema, to default to null values. This allows for normal event parsing to our Kafka Cluster even if any fields in our JSON input file are missing. While this configuration offers maximum flexibility, it does not check our data before it is published to our application.

2. In the second scenario, we select a few or just one value to default to null. For instance, we could set our temperature field to null in our schema if a group of sensors or devices doesn't support this measurement. In this instance, messages would still be parsed even if the temperature field was missing from the input messages.

3. The third scenario involves not setting any field to default at null. This enforces strict data validation, requiring all our fields to be included in our input file.

4. The fourth case involves disallowing additional fields beyond those included in the schema fields. By default, the schema registry permits extra fields if the required ones have been validated. For example, if an incoming message includes an additional 'pressure' field, we can decide whether or not we want this event to be produced.

In the four scenarios outlined, our goal was to showcase the versatility of our schema registry implementation. As a result, we ranged from a highly flexible data validation routine to a very strict one. The degree of flexibility is dependent on the requirements of our application and implementation at each instance. This underscores the importance of maintaining a balance between adaptability of data and its consistency.

5.2 Event Size Reduction

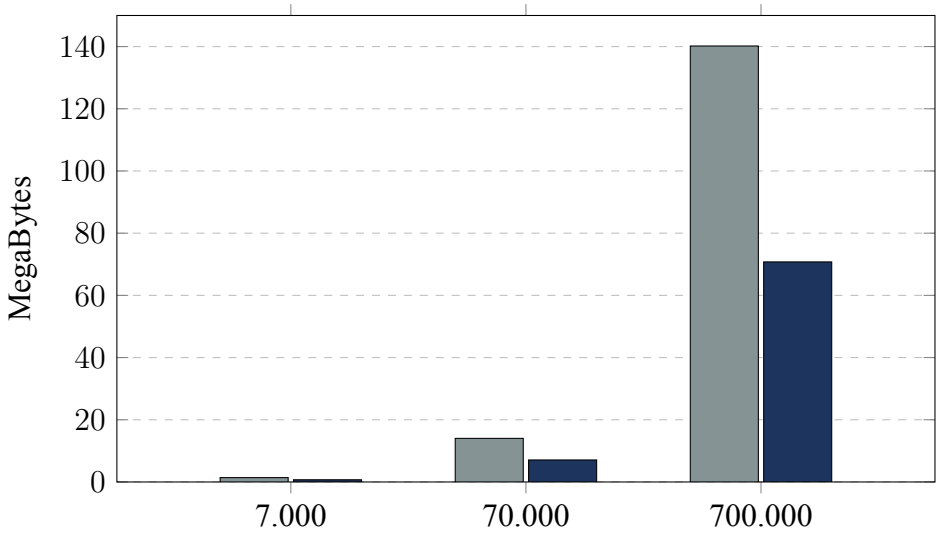

Table 5.1: JSON to Avro size of messages comparison

Our Schema Registry implementation enables us to store events in our cluster in a more compact and efficient manner. We produced a set of messages to observe the size reduction between native or string JSON format messages, and Avro serialized ones. Our data sets are in groups of 7,000, 70,000, and 700,000. As presented in table 5.1, the size reduction ranges from 1,402 KB to 707 KB, 14.02 MB to 7.076 MB, and 140.19 MB to 70.76 MB, for the batched 7,000, 70,000 and 700,000 messages respectively. Therefore, the reduction rate for our input message when converted to Avro format is $\approx$ 48-50%. This size reduction can vary based on the structure of the original fields and the initial event size, and it can be

increased even further. Although the size reduction in our case is in the range of megabytes, it is clear that this sets the stage for larger scale implementations where tens or even hundreds of millions of events are constantly stored in the clusters.

5.3 Throughput Testing

We have created a realistic deployment scenario to test whether our implementation can handle the expected payload. In this scenario, we emulate a situation with a hundred devices producing data, setting up a hundred producers that constantly send data to our Kafka cluster. However, in our tests, we produce messages every second and every five minutes. This dual frequency approach provides a clear view of the volume of messages our implementation can handle, both with our standard deployment frequency, and with a significantly higher one. Often, benchmarks test throughput in unrealistic environments. By conducting realistic tests, we can confirm that our implementation can handle the load during production. We leverage Kafka's partition mechanism in this process, with each of the hundred devices representing a hundred partitions. This is an optimal use case where each partition is connected to a single device.

Given that our production message frequency is every five minutes, by sending messages every second in our cluster, we emulate the behaviour of having a significantly larger number of devices publishing messages. It is also important to note that throughput results often depend on the event message size. Therefore, we conduct our experiment using our application's message structure. The experimentation was carried out on an i7-6700k system, utilizing Docker engine hosted on WSL2, with an allocation of 6 logical cores. Our implementation was able to handle incoming messages at frequencies of both every second and every five minutes without any downtime. Similarly, our consumer was able to read data without any downtime, even during the test with a frequency of every second. As a result, we have verified that our cluster implementation can handle a hundred messages being sent every second from a hundred different devices, based on our given system configuration.

Chapter 6

Conclusions

In this thesis, we developed a full-stack air pollution monitoring system adhering to IoT fundamentals, but also combined with capabilities found in enterprise cloud services. Our key goals and challenges were to extend the capabilities of the SYNAISTHISI platform with advanced event streaming services, create a live environmental air quality monitoring application and make our platform as scalable and interconnected as possible to host other similar applications.

We analyzed our motivations, considering the need to create air pollution monitoring applications and infrastructures accessible to the public. We created an event-driven middleware capable of interconnecting with most of the established protocols in the IoT industry. We configured our Kafka cluster in a containerized, multi-broker implementation and adopted the latest control plane KRaft, for managing our Kafka nodes. Furthermore, we interconnected our primary cluster with Kafka Connect and its external communication protocols like AMQP and MQTT, showcasing that almost any other supported external system, service, or database can be connected as well. We added a Schema Registry to our data flow, making our application schema-enabled and providing us with features of advanced data validation, integrity and controlled data evolution. We developed a back-end service that fetches our data from our Kafka Cluster in real-time, eliminating the need for an additional database for our live event messages streaming. We facilitated a Web-Sockets application over a TCP channel, to parse the consumed data to our front-end map, efficiently matching the high throughput requirements of our platform. Last but not least, we created a React-based live map utilizing Leaflet maps, where each device and sensor is automatically added as a point on the map. Each sensor is updated on our live map and a dashboard of tables for each corresponding

sensor provides us with historical measurements for up to a week.

By utilizing and orchestrating so many different services, we focused on making them as independent and modular as possible. This way, our application remains highly scalable, accessible and configurable. The whole stack is constructed and presented in a way that showcases each layer, highlighting the re-usability and adoption of our platform in other respective projects. Additionally, each individual service was chosen with the aspect of belonging to an open-source project that is being maintained, further future-proofing our platform. Features such as fault tolerance, data validation, high throughput and data integrity, which are typically found in paid enterprise cloud solutions, were incorporated into our platform. This was achieved by carefully selecting the appropriate open-source projects and services and then optimizing and configuring them to meet our requirements. Therefore, we developed a platform that is accessible to everyone and aligns with the key IoT and open-source principles, while at the same time incorporating advanced cloud features and fundamentals.

6.1 Future work

Regarding future work, there are several improvements and upgrades that could be applied to our implementation. Initially the development of enterprise-level security by utilizing SASL_SSL and TLS(SSL) security protocols. Our configuration would involve using SSL for internal listeners(clients) and inter-broker communications. Additionally, external listeners(clients) could be configured to utilize SASL_SSL, integrated with a Kerberos server through the GSSAPI authentication mechanism. Such integration necessitates thorough research and testing based on the cluster's needs and priorities, in order to achieve the optimal balance between security and performance. Additionally, we can enhance the default behavior of Kafka's key mapping mechanism. This could be achieved by integrating an external load balancing service or by developing and configuring our own key hashing mechanism, to distribute the events across the partitions equally and efficiently.

Bibliography

[1] Somayya Madakam, Vihar Lake, Vihar Lake, Vihar Lake, et al. Internet of things (iot): A literature review. *Journal of Computer and Communications*, 3(05):164, 2015. [Ref].

[2] Mohammed Shamim Kaiser, Khin T. Lwin, Mufti Mahmud, Donya Hajializadeh, Tawee Chaipimonplin, Ahmed Sarhan, and Mohammed Alamgir Hossain. Advances in crowd analysis for urban applications through urban event detection. *IEEE Transactions on Intelligent Transportation Systems*, 19(10):3092–3112, 2018. [Ref].

[3] Gaetano Rocco, Claudia Pipino, and Claudio Pagano. An overview of urban mobility: Revolutionizing with innovative smart parking systems. *Sustainability*, 15(17), 2023. [Ref].

[4] Karen Rose, Scott Eldridge, and Lyman Chapin. The internet of things: An overview. *The internet society (ISOC)*, 80:1–50, 2015.

[5] Parvaneh Asghari, Amir Masoud Rahmani, and Hamid Haj Seyyed Javadi. Internet of things applications: A systematic review. *Computer Networks*, 148:241–261, 2019. [Ref].

[6] Simon Elias Bibri. The iot for smart sustainable cities of the future: An analytical framework for sensor-based big data applications for environmental sustainability. *Sustainable Cities and Society*, 38:230–253, 2018. [Ref].

[7] Shiv. H. Sutar, Rohan Koul, and Rajani Suryavanshi. Integration of smart phone and iot for development of smart public transportation system. In *2016 International Conference on Internet of Things and Applications (IOTA)*, pages 73–78, 2016. [Ref].

[8] Charles Harper and Monica Snowden. *Environment and society: Human perspectives on environmental issues*. Routledge, 2017.

[9] Industrial pollutant releases to air in europe. [Ref]. Ημερομηνία πρόσβασης: 06-01-2024.

[10] European environment agency footprint. [Ref]. Ημερομηνία πρόσβασης: 06-01-2024.

[11] Swati Dhingra, Rajasekhara Babu Madda, Amir H. Gandomi, Rizwan Patan, and Mahmoud Daneshmand. Internet of things mobile–air pollution monitoring system (iot-mobair). *IEEE Internet of Things Journal*, 6(3):5577–5584, 2019. [Ref].

[12] Charilaos Akasiadis, Vassilis Pitsilis, and Constantine D. Spyropoulos. A multi-protocol iot platform based on open-source frameworks. *Sensors*, 19(19), 2019. [Ref].

[13] Massimo Stafoggia, Paola Michelozzi, Alexandra Schneider, Ben Armstrong, Matteo Scortichini, et al. Joint effect of heat and air pollution on mortality in 620 cities of 36 countries. *Environment International*, 181:108258, 2023. [Ref].

[14] Prakash Thangavel, Duckshin Park, and Young-Chul Lee. Recent insights into particulate matter (pm2.5)-mediated toxicity in humans: An overview. *International Journal of Environmental Research and Public Health*, 19(12), 2022. [Ref].

[15] Michele De Donno, Koen Tange, and Nicola Dragoni. Foundations and evolution of modern computing paradigms: Cloud, iot, edge, and fog. *IEEE Access*, 7:150936–150948, 2019. [Ref].

[16] Apache kafka and its origin. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[17] Iq air. [Ref]. Ημερομηνία πρόσβασης: 05-01-2024.

[18] Air pollution control. [Ref]. Ημερομηνία πρόσβασης: 05-01-2024.

[19] Ghazanfar Latif, Jaafar M. Alghazo, R. Maheswar, P. Jayarajan, and A. Sampathkumar. *Impact of IoT-Based Smart Cities on Human Daily Life*, pages 103–114. Springer International Publishing, Cham, 2020. [Ref].

[20] B.B. Gupta and Megha Quamara. An overview of internet of things (iot): Architectural aspects, challenges, and protocols. *Concurrency and Computation: Practice and Experience*, 32(21):e4946, 2020. e4946 CPE-18-0159.R1[Ref].

[21] Alessio Botta, Walter de Donato, Valerio Persico, and Antonio Pescapé. Integration of cloud computing and internet of things: A survey. *Future Generation Computer Systems*, 56:684–700, 2016. [Ref].

[22] Ciro Formisano, Daniele Pavia, Levent Gurgen, Takuro Yonezawa, Josè Antonio Galache, Keiko Doguchi, and Isabel Matranga. The advantages of iot and cloud applied to smart cities. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 325–332, 2015. [Ref].

[23] Greenyourair. [Ref]. Ημερομηνία πρόσβασης: 08-01-2024.

[24] Benefits of open-source software for developers, managers and business. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[25] Open source contributor index. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[26] Open.intel. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[27] Event streaming platform. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[28] Apache kafka contributors. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[29] Amazon kinesis. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[30] Azure event hubs. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[31] Google pub/sub. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[32] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-native applications. *IEEE Cloud Computing*, 4(5):16–21, 2017. [Ref].

[33] Yohanes Yohanie Fridelin Panduman, Nobuo Funabiki, Pradini Puspitaningayu, Minoru Kuribayashi, Sritrusta Sukaridhoto, and Wen-Chung Kao. Design and implementation of semar iot server platform with applications. *Sensors*, 22(17), 2022. [Ref].

[34] Yohanes Yohanie Fridelin Panduman, MR Ulil Albaab, AR Anom Besari, Sritrusta Sukaridhoto, Anang Tjahjono, and RP Nourma Budiarti. Implementation of data abstraction layer using kafka on semar platform for air quality monitoring. *International Journal on Advanced Science, Engineering and Information Technology*, 9(5):1520, 2019. [Ref].

[35] Joschka Kersting, Michaela Geierhos, Hanmin Jung, and Taehong Kim. Internet of things architecture for handling stream air pollution data. In *IoTBDS*, pages 117–124, 2017. [Ref].

[36] Sami Kaivonen and Edith C.-H. Ngai. Real-time air pollution monitoring with sensors on city bus. *Digital Communications and Networks*, 6(1):23–30, 2020. [Ref].

[37] Ihsane Gryech, Yassine Ben-Aboud, Bassma Guermah, Nada Sbihi, Mounir Ghogho, and Abdellatif Kobbane. Moreair: A low-cost urban air pollution monitoring system. *Sensors*, 20(4), 2020. [Ref].

[38] Lidia Morawska, Phong K. Thai, Xiaoting Liu, Akwasi Asumadu-Sakyi, and et al. Applications of low-cost sensing technologies for air quality monitoring and exposure assessment: How far have they gone? *Environment International*, 116:286–299, 2018. [Ref].

[39] Docker. [Ref]. Ημερομηνία πρόσβασης: 04-01-2024.

[40] Docker network. [Ref]. Ημερομηνία πρόσβασης: 04-01-2024.

[41] Docker compose. [Ref]. Ημερομηνία πρόσβασης: 04-01-2024.

[42] Avro. [Ref]. Ημερομηνία πρόσβασης: 10-01-2024.

[43] Multi cluster implementation. [Ref]. Ημερομηνία πρόσβασης: 10-01-2024.

[44] Apache zookeeper. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[45] Apache kafka kraft. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[46] Kraft recommendation. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[47] Rabbitmq introduction. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[48] Mqtt introduction. [Ref]. Ημερομηνία πρόσβασης: 09-01-2024.

[49] Kafka connect. [Ref]. Ημερομηνία πρόσβασης: 10-01-2024.

[50] Schema registry. [Ref]. Ημερομηνία πρόσβασης: 10-01-2024.

[51] Akhq control panel. [Ref]. Ημερομηνία πρόσβασης: 13-01-2024.

[52] React js. [Ref]. Ημερομηνία πρόσβασης: 13-01-2024.

[53] Open street map. [Ref]. Ημερομηνία πρόσβασης: 13-01-2024.