Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

A LoRaWAN-Based Monitoring System for a Smart Campus

João Rodrigo Caiado Condeço

Orientador(es) | Pedro Salgueiro

Vitor Beires Nogueira

Évora 2026

Universidade de Évora - Escola de Ciências e Tecnologia

Mestrado em Engenharia Informática

Dissertação

A LoRaWAN-Based Monitoring System for a Smart Campus

João Rodrigo Caiado Condeço

Orientador(es) | Pedro Salgueiro

Vitor Beires Nogueira

Évora 2026

A dissertação foi objeto de apreciação e discussão pública pelo seguinte júri nomeado pelo Diretor da Escola de Ciências e Tecnologia:

Presidente	|	Teresa Gonçalves (Universidade de Évora)
Vogais	|	João L. M. Pereira (Universidade de Évora) (Arguente) Pedro Salgueiro (Universidade de Évora) (Orientador)

Évora 2026

To my father, who by working late into the night in our garage, unknowingly showed me the path I wanted to follow.

And to Margarida, my girlfriend, for always encouraging me to move forward, never backward.

Acknowledgements

I am deeply grateful to my girlfriend, Margarida, for her unconditional belief in my abilities and her daily support through the good and bad moments alike. During the difficult situations that arose throughout the elaboration of this dissertation, she was my pillar.

I must mention and dedicate an immeasurable thank you to my Father who, beyond being my inspiration from a very young age to pursue this field, has always done everything in his power to enable me to reach my goals. It was watching him work late into the night in our garage that I decided I wanted to become a computer scientist.

I thank my work colleagues Fernando and David for the knowledge, teachings, and support they shared with me, and my colleague Bernardo who accompanied me on this journey both at university and professionally.

A special thank you to Professor Pedro Salgueiro for always being available to help me from the start of my undergraduate degree to this day.

And finally, but no less importantly, I also thank all the support from my family and close friends.

Contents

List of Figures

List of Tables

Acronyms

ABP	Activation By Personalization
ADR	Adaptive Data Rate
AES	Advanced Encryption Standard
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
BLE	Bluetooth Low Energy
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
DevEUI	Device Extended Unique Identifier
FHSS	Frequency Hopping Spread Spectrum
gRPC	Google Remote Procedure Call
HTTP	HyperText Transfer Protocol
HVAC	Heating, Ventilation, and Air Conditioning
ICT	Information and Communication Technologies
IoT	Internet of Things
ISM	Industrial, Scientific and Medical
JSON	JavaScript Object Notation
LPWAN	Low Power Wide Area Network
LoRa	Long Range
LoRaWAN	Long Range Wide Area Network
MAC	Media Access Control
MQTT	Message Queuing Telemetry Transport
NB-IoT	Narrowband Internet of Things
OLAP	Online Analytical Processing
OTAA	Over-The-Air Activation
PHY	Physical Layer
QoS	Quality of Service
RBAC	Role-Based Access Control

REST	Representational State Transfer
RF	Radio Frequency
RSSI	Received Signal Strength Indicator
SF	Spreading Factor
SIMD	Single Instruction, Multiple Data
SNR	Signal-to-Noise Ratio
SQL	Structured Query Language
TLS	Transport Layer Security
VOC	Volatile Organic Compound

Abstract

The proliferation of Internet of Things in recent years has created new opportunities for the development of smart environments that are capable of performing continuous monitoring, supporting data-driven decision-making, and optimising the use of physical resources across a wide range of application domains. University campuses represent a compelling context for the deployment of such systems, given the diversity of buildings and spaces that must be managed, the institutional interest in sustainability and efficiency, and the availability of technical expertise. This dissertation presents the design, implementation, and evaluation of a LoRaWAN-based IoT monitoring system intended for smart campus environments, with a particular emphasis on creating a platform that is scalable, modular, and built entirely upon open-source technologies in order to avoid the recurring costs and vendor dependencies that are associated with commercial IoT platforms.

The proposed system architecture addresses several of the key challenges that arise in large-scale IoT deployments, including the heterogeneity of sensor devices, the need to integrate data from multiple sources into a coherent processing pipeline, the requirement for horizontal scalability as the number of deployed devices grows, and the expectation of near-real-time data availability for monitoring and alerting purposes. LoRaWAN was selected as the communication technology, employing ChirpStack as the network server, a custom protocol bridge implemented in Rust for data transformation and routing, and ClickHouse as the analytical database, with each component was selected to address the specific operational constraints and performance requirements of the campus deployment context.

Keywords: Internet of Things, LoRaWAN, Smart Campus, MQTT, Microservices, Environmental Monitoring, RabbitMQ, Rust

Resumo

Um Sistema de Monitorização Baseado em LoRaWAN para um Campus Inteligente

A proliferação da Internet das Coisas nos últimos anos tem criado novas oportunidades para o desenvolvimento de ambientes inteligentes capazes de realizar uma monitorização contínua, apoiar a tomada de decisões baseada em dados e otimizar a utilização de recursos físicos numa vasta gama de domínios de aplicação. Os campus universitários representam um contexto ideal para a implementação deste tipo de sistemas, dada a diversidade de edifícios e espaços que necessitam de ser geridos, o interesse institucional na sustentabilidade e eficiência, e a disponibilidade de conhecimentos técnicos. Esta dissertação apresenta a conceção, implementação e avaliação de um sistema de monitorização IoT baseado em LoRaWAN, destinado a ambientes de campus inteligentes. É dado um ênfase particular à criação de uma plataforma que seja escalável, modular e construída inteiramente com base em tecnologias de código aberto, com o intuito de evitar os custos recorrentes e as dependências de fornecedores associadas às plataformas comerciais de IoT.

A arquitetura de sistema proposta aborda vários dos principais desafios que surgem em implementações IoT de grande escala, incluindo a heterogeneidade dos dispositivos sensores, a necessidade de integrar dados de múltiplas fontes num fluxo de processamento coerente, o requisito de escalabilidade horizontal à medida que o número de dispositivos instalados cresce, e a expectativa de disponibilidade de dados quase em tempo real para fins de monitorização e alerta. LoRaWAN foi a tecnologia de comunicação selecionada, utilizando o ChirpStack como servidor de rede, uma ponte de protocolos (protocol bridge) desenvolvida à medida em Rust para a transformação e encaminhamento de dados, e o ClickHouse como base de dados analítica. Cada um destes componentes foi selecionado com o intuito de responder às restrições operacionais específicas e aos requisitos de desempenho do contexto de implementação no campus.

Palavras-chave: Internet das Coisas, LoRaWAN, Campus Inteligente, MQTT, Microsserviços, Monitorização Ambiental, RabbitMQ, Rust

Chapter 1

Introduction

This chapter introduces the work undertaken in this dissertation, outlining the motivation, core objectives, and practical applications within a real-world context.

1.1 Motivation and Context

Over the past decade, the digital transformation of educational institutions has accelerated significantly. Driven by advances in connectivity, improved sensing hardware, and more sophisticated data analytics, universities are increasingly adopting Internet of Things (IoT) technologies to create "smart campus" environments. The foundational concept involves interconnected sensors and intelligent systems working together to monitor campus conditions, such as energy usage, environmental quality, space utilization, and security [Zanella et al., 2014]. Consequently, a campus capable of sensing and reacting to its environment becomes more sustainable, safe, and efficient.

Despite the clear benefits, implementing these systems at scale presents significant technical hurdles. Educational facilities are often spread over large areas; the University of Évora campus, for instance, spans a considerable area in the Alentejo region, comprising a complex mix of 16th-century historical buildings and modern structures. This physical spread requires communication technologies capable of covering several kilometers reliably and efficiently, particularly because the sensor nodes are battery powered and widely distributed [Domínguez-Bolaño et al., 2024]. Furthermore, smart campus ecosystems inherently involve heterogeneous devices from various vendors, each using distinct communication protocols and data formats. Cohesive integration of these disparate components is a complex undertaking.

Short-range wireless technologies, such as Wi-Fi, Bluetooth, and Zigbee, can support high data rates but possess prohibitive limitations for campus-wide deployments. Their limited range, generally under a hundred metres, dictates that covering an entire campus would require a dense, expensive deployment of access points or re-

peaters, leading to frequent battery replacements and high power consumption [Raza et al., 2017]. When deploying hundreds of sensors across buildings, courtyards, and parking areas, these limitations become critical barriers to adoption.

This is where Low Power Wide Area Network (LPWAN) technologies become essential. Protocols like Long Range Wide Area Network (LoRaWAN) were specifically designed for these scenarios, offering long-range communication often covering several kilometres with a single gateway and exceptionally low power consumption. A LoRaWAN sensor node can operate on a small battery for five to ten years, fundamentally changing deployment strategies. For a university, this means sensors can be placed on rooftops, inside storage rooms, or along garden paths, where running power cables or performing regular battery maintenance is impractical [Augustin et al., 2016].

However, establishing a reliable wireless network is only the first step. Once sensor data is transmitted, it requires systematic collection, processing, and storage to be made available to downstream users and systems. Building a robust data pipeline capable of handling continuous streams from thousands of devices while simultaneously supporting real-time alerting and historical analysis requires critical architectural decisions. These decisions have long-term consequences for the maintainability, scalability, and overall utility of the system.

Within this context sits the present dissertation. The research originated from a concrete institutional need at the Évora University to develop a monitoring platform for campus operations and research activities. Rather than adopting a commercial, closed-source solution, the objective was to design and build a system from the ground up using open-source technologies. This approach was chosen not only for cost efficiency but, more importantly, to maintain full control over the architecture and data, resulting in a flexible system capable of evolving alongside institutional needs.

1.2 Problem Statement

Deploying an IoT monitoring system for a smart campus introduces several interconnected technical challenges. This dissertation aims to systematically address these core issues.

The first challenge is **communication infrastructure**. Campus environments require connectivity that balances long-range coverage, low power consumption, and adequate data capacity. Because short-range technologies demand excessive infrastructure and maintenance, a unified wireless layer capable of comprehensive indoor and outdoor coverage is essential.

The second challenge involves **data integration and interoperability**. A comprehensive IoT deployment inevitably encompasses diverse sensor types, communication protocols, and data formats. The system must establish a unified data pipeline

that integrates these heterogeneous devices while maintaining consistent data quality. This includes resolving how LoRaWAN network servers publish data and how downstream services expect to consume it.

Scalability and performance represent another critical concern. As a deployment scales from a pilot phase of a few sensors to a campus-wide network of thousands, the backend architecture must adapt. The system needs to handle continuous data ingestion without performance degradation, support real-time processing for immediate alerting, and facilitate the efficient querying of historical data spanning months or years.

Furthermore, **system modularity and maintainability** must be addressed. Monolithic designs, while simpler to initiate, often create tight couplings that make individual components difficult to update, replace, or scale independently. To ensure long-term viability, the architecture requires a modular approach where distinct services can be deployed and maintained autonomously.

Finally, a significant challenge lies in **selecting the appropriate messaging infrastructure**. While the initial architecture utilized Apache Kafka for event streaming, its operational complexity prompted a reevaluation of lighter alternatives, such as RabbitMQ. Balancing operational capability against architectural complexity remained a recurring theme throughout the development process.

1.3 Research Objectives

To address these challenges, this dissertation establishes five primary research objectives:

First, to conduct a comprehensive analysis of state-of-the-art IoT communication technologies, with a specific focus on LPWAN solutions and their applicability to smart campus environments. This analysis forms the basis for selecting and justifying LoRaWAN as the underlying communication layer.

Second, design a modular scalable IoT platform architecture that integrates LoRaWAN connectivity with modern distributed systems technologies, including message brokers, protocol bridges, and time-series analytics databases.

Third, to document the architectural evolution of the system from its initial design, based on Apache Kafka, to a revised architecture utilizing RabbitMQ and high-performance Rust middleware. This objective includes detailing the rationale behind each architectural pivot to assist future implementations.

Fourth, to implement and validate a functional prototype that demonstrates the feasibility of the proposed architecture and lays the groundwork for a full campus-wide deployment.

Fifth, to provide practical deployment guidelines and insights for other institutions

undertaking similar projects, identifying potential pitfalls and sharing integration strategies.

1.4 Contributions

This research provides several notable contributions to the field of smart campus IoT deployments:

The primary contribution is an **end-to-end architectural framework**. This microservices-based architecture addresses scalability, modularity, and performance across the entire data lifecycle from LoRaWAN sensors and protocol bridging to message streaming, analytical storage, and user-facing applications.

The second contribution is a custom **protocol bridge design developed in Rust**. This component translates between ChirpStack Message Queuing Telemetry Transport (MQTT)-based data output and the designated message broker. Rust was selected due to its high performance, memory safety, and low resource consumption.

Third, the dissertation provides thorough **architectural evolution documentation**. A detailed comparison is presented between the initial Apache Kafka-based architecture and the subsequent iteration utilizing RabbitMQ with Rust middleware, encompassing the underlying reasoning, trade-offs, and practical implications of these shifts.

Fourth, a comprehensive **comparison of LPWAN technologies** is offered. This evaluates LoRaWAN, NB-IoT, and Sigfox strictly within the context of smart campus requirements, providing a well-grounded justification for the selection of LoRaWAN.

Finally, the work delivers **practical deployment guidance**, synthesizing lessons learned, technology selection criteria, and component integration strategies to assist other institutions planning large-scale IoT initiatives.

1.5 Dissertation Structure

The remainder of this dissertation is organized as follows:

Chapter 2 reviews the state of the art in IoT technologies relevant to smart campus deployments. It covers fundamental concepts, communication protocols (emphasizing LPWAN), smart city frameworks, and IoT platform architectures, including message brokers, databases, and network servers.

Chapter 3 details the proposed system architecture. It describes both the initial design (Version 1, utilizing Apache Kafka) and the revised design (Version 2, utilizing RabbitMQ and Rust middleware), clearly articulating the design principles and motivations driving the system's evolution.

Chapter 4 discusses the implementation specifics, including the development of the MQTT bridge in Rust, ChirpStack configuration, message broker setup, and the ClickHouse database schema. It also outlines the microservices that comprise the application layer.

Chapter 5 evaluates the system, presenting performance observations, a qualitative assessment of the architecture against the initial requirements, and a validation of its operational functionality.

Chapter 6 summarizes the developed work, discusses identified limitations, and outlines potential directions for future research and continuous improvement.

1.6 Generative AI as a Tool Statement

To refine the quality of this manuscript, I declare the use of Generative Artificial Intelligence tools during the development of this work, specifically to refine sentence structure, research relevant literature, and clarify technical concepts. All AI-generated content was rigorously reviewed, edited, and validated by the author, who assumes full responsibility and authorship for the final text.

1.7 Acknowledgments

This work was carried out within the scope of the project “TID4AGRO - Tecnologias Avançadas, Inovadoras e Digitais para o sector agroalimentar da EUROACE”, co-financed by the INTERREG VI-A Spain - Portugal Programme (POCTEP) 2021-2027 of the European Commission, project number 0100_TID4AGRO_4_E. The LoRaWAN sensors and gateway utilized in this research were acquired with the support of this project, representing a foundational contribution to this dissertation.

Chapter 2

State of the Art

This chapter provides the theoretical and technological foundations upon which the work presented in this dissertation is built. It covers the core concepts, established technologies, and prior research that are relevant to the design and implementation of an IoT monitoring system for a smart campus environment. I begin by introducing fundamental IoT concepts and the broader contexts of smart cities and smart campuses, after which I review the communication technologies commonly employed in IoT deployments, with particular emphasis on LPWAN protocols. The chapter concludes with an overview of the components that make up a complete IoT platform, including network servers, message brokers, databases, and the communication protocols that tie them together.

2.1 IoT Introduction and Smart Cities

The Internet of Things represents a fundamental shift in which physical objects are augmented with sensing, processing, and communication capabilities, allowing them to interact with each other through large networks and exchange data with information systems. In essence, IoT can be understood as an extension of the traditional Internet into the physical world, where everyday objects become active participants in digital processes. Ab Rahman [2015] pointed out that IoT was conceived in order to provide connectivity "for anything, at anytime and anywhere," bridging physical environments with digital systems and enabling entirely new types of services and applications.

2.1.1 Fundamental Internet of Things Concepts

At a conceptual level, an IoT system is typically structured around three fundamental layers: the perception layer, the network layer, and the application layer [Al-Fuqaha et al., 2015]. Each of these layers addresses a distinct set of responsibilities, and the design choices made at each level have implications that propagate through-

out the entire system.

The perception layer serves as the interface between the physical and digital realms, encompassing sensors and actuators that interact with the physical environment to gather measurements and subsequently transform them into digital data. This layer's scope of monitoring is broad, potentially encompassing temperature, humidity, air quality, light intensity, motion, and energy consumption, among other variables. In the context of campus deployments, the perception layer is often characterized by its diversity, given the wide spectrum of monitoring requirements, which can range from basic temperature and humidity probes to more sophisticated air quality analyzers that assess CO2 concentration and particulate matter [Zanella et al., 2014]. The selection of specific sensors and their placement are significantly influenced by the communication technology employed, as both power availability and the proximity to the nearest gateway directly determine the feasibility of deploying sensors and their optimal locations.

The network layer provides the connectivity between devices and backend systems, ensuring that data collected at the perception layer can be reliably transported to processing and storage infrastructures. There is a wide range of communication technologies available for this purpose, and they are distinguished from each other by characteristics such as coverage, data rate, power consumption, and scalability. Short-range technologies like ZigBee and Bluetooth Low Energy are designed for low-power communication over small areas, while Wi-Fi offers considerably higher throughput but at the expense of energy efficiency. Research consistently shows that the choice of communication protocol significantly affects system performance, power use, and scalability [Ab Rahman, 2015]. These factors are crucial when designing a large-scale deployment. For smart campus scenarios that cover wide geographical areas and require long battery lifetimes, low-power wide-area protocols such as LoRaWAN address limitations that traditional short-range solutions are not able to overcome.

The application layer sits on top and is responsible for transforming raw sensor data into something that can actually be used, encompassing data processing, analysis, visualization, and integration with end-user services. Whether it is a dashboard showing current environmental conditions, an alert triggered by unusual sensor readings, or a trend analysis spanning several months, the application layer is what delivers value to the people and systems that consume the data. Modern IoT platforms increasingly rely on cloud and edge computing paradigms in order to achieve scalability and keep latency at acceptable levels, and the adoption of microservices architectures at this layer allows applications to become modular, resilient, and easier to evolve over time. This is particularly relevant in campus settings where requirements change frequently as new use cases emerge [Ortiz et al., 2022].

Beyond these three layers, there are several cross-cutting concerns that shape design decisions across the entire IoT stack and that deserve attention.

Energy efficiency is arguably the single most important constraint in many IoT

deployments. The majority of IoT devices operate on batteries and are expected to function for years without maintenance or battery replacement, which means that low-power communication is not merely a desirable feature but rather an essential requirement. Research on low-power networks has consistently emphasized that minimizing communication overhead and optimizing data transmission patterns are key factors in extending device lifetime, particularly in large-scale sensor networks deployed for monitoring purposes [Sanchez-Iborra and Cano, 2016].

Security and privacy are concerns that span every layer of an IoT system. Given that these systems involve a wide variety of devices using different protocols and data formats, it is necessary to handle authentication, data integrity, and confidentiality at every level. This becomes even more critical in campus environments where the system might be tracking occupancy data or access control information alongside environmental readings, requiring device authentication, encrypted communications, secure firmware updates, and protection against various types of attacks [Roman et al., 2013].

Interoperability is a persistent challenge in IoT ecosystems, which inherently mix devices from different manufacturers that use different protocols and data formats. Getting heterogeneous components to work together is essential for any deployment that aims to be sustainable in the long term, as one wants to avoid being locked into a single vendor and to ensure that the deployment can evolve as new devices and technologies become available [Noura et al., 2019].

Scalability affects the entire architecture of an IoT system. A campus deployment might start as a pilot with just a few dozen sensors, but it could eventually grow to include hundreds or even thousands of devices. The system needs to be designed from the beginning with this potential growth in mind, because retrofitting scalability into an architecture that was not built for it is significantly more difficult and costly than incorporating it from the start [Khan et al., 2012].

2.1.2 Smart Cities Paradigm

The smart city concept represents a vision in which information and communication technologies are employed in order to improve urban life, enhance the efficiency of city operations, and push toward environmental sustainability. Smart cities bring together IoT sensors, data analytics, and intelligent systems across multiple domains including transportation, energy management, public safety, and environmental monitoring [Zanella et al., 2014]. The fundamental idea is to instrument the urban environment with sensors and connect them to data processing infrastructure, thereby providing city administrators and citizens with real-time visibility into what is actually happening and enabling better-informed decisions.

In practice, smart city implementations typically involve sensor networks that monitor air quality and traffic conditions, analytics platforms that identify patterns and detect anomalies, and applications that provide citizens with real-time information

about public services. Projects carried out in various cities around the world have demonstrated tangible benefits, including reduced energy consumption, improved traffic management, enhanced public safety through predictive analytics, and decreased pollution levels [Fortes et al., 2019]. Having this in mind, it should also be noted that smart city deployments are not without challenges, as integrating heterogeneous systems is inherently complex, deploying and maintaining sensor infrastructure at scale requires significant investment, and widespread monitoring raises privacy concerns that require careful consideration.

2.1.3 Smart Campus Concept

A smart campus applies the principles and technologies of smart cities to universities and colleges. The goal is to create connected educational environments that improve learning, make operations more efficient, and support advanced research. Smart campuses are similar to smart cities, but they have the advantage of being in controlled settings. This allows for experimentation and gives the institution direct control over its infrastructure [Villegas-Ch et al., 2019].

Several application areas are particularly valuable in the context of smart campuses. Environmental monitoring involves deploying sensors to measure indoor and outdoor conditions such as temperature, humidity, CO_2 levels, volatile organic compounds, and particulate matter, providing data that supports healthy learning environments, helps identify air quality problems, and indicates whether heating, ventilation, and air conditioning systems are performing adequately. Outdoor monitoring can additionally track weather patterns and pollution levels, supporting environmental research and campus sustainability initiatives [Spachos, 2020].

Energy management through IoT allows institutions to monitor electricity consumption at the building level and sometimes even down to individual sub-meters. Smart HVAC systems can adjust their operation based on actual occupancy and environmental conditions rather than following fixed schedules, and pilot studies have documented energy savings in the range of twenty to forty percent [Fortes et al., 2019]. Space utilization monitoring, which relies on occupancy sensors and people-counting technologies, provides insights into how classrooms, laboratories, and common areas are actually being used, thereby enabling the optimization of class scheduling and the identification of underutilized spaces.

Campus safety and security also benefit from connected systems, as environmental hazard detectors, access control mechanisms, and emergency alert systems can share information with each other to enable faster incident response. Research infrastructure is perhaps the least immediately obvious application but may be the most uniquely valuable one for universities, since a smart campus platform becomes a real-world IoT testbed where students and researchers can experiment with wireless networking, data science, and artificial intelligence under authentic conditions [Muhamad et al., 2017].

Research on smart campus deployments has carefully documented both the benefits and the challenges involved. Issues related to interoperability, scalability, and data management need to be thought through during the design phase rather than addressed retroactively, as attempting to fix fundamental architectural shortcomings after deployment is considerably more difficult [Domínguez-Bolaño et al., 2024].

2.2 Communication Technologies for IoT

2.2.1 Overview of IoT Communication Technologies

The IoT encompasses a wide range of communication protocols that enable heterogeneous devices to share information between them, but even though they are built for a broadly similar purpose, these protocols differ substantially in terms of transmission range, power consumption, data rate, and cost. Having this in mind, IoT networks are typically categorised into two main classes: short-range and long-range communication technologies [Ab Rahman, 2015].

The choice of communication technology for a given deployment is not purely a technical decision, since it directly affects infrastructure costs, maintenance requirements, battery replacement frequency, and the kinds of applications that are feasible. Understanding the trade-offs between the available options is therefore essential when making design decisions for any IoT system.

2.2.2 Short-Range Technologies

Short-range wireless technologies are well established and primarily support applications that require either high throughput or low latency within limited spatial areas. For IoT deployments, three technologies are most commonly considered: Wi-Fi, Bluetooth Low Energy, and ZigBee.

Wi-Fi, based on the IEEE 802.11 standard, delivers high data rates, with modern standards like 802.11ac, 802.11ax, and 802.11be achieving multi-gigabit throughput, which makes Wi-Fi the obvious choice for any device that needs to transfer large amounts of data. However, Wi-Fi consumes relatively high power compared to other IoT protocols, and this significant difference makes it unsuitable for battery-powered sensors that need to operate independently for years. In IoT deployments, Wi-Fi is therefore typically reserved for gateways or other powered infrastructure nodes [Hiertz et al., 2010].

Bluetooth Low Energy, which was introduced with the Bluetooth 4.0 specification, was purpose-built for ultra-low-power applications and uses adaptive frequency hopping in order to reduce interference and optimize energy consumption. Devices running BLE can operate for months or even years on small batteries, though the data rate is limited to approximately 1 Mbps and the range extends only to a few

tens of metres. Gomez et al. [2012] note that BLE is particularly well suited for personal area networks, wearable devices, and indoor positioning applications, and later Bluetooth versions have extended the range while maintaining the low-power characteristics.

ZigBee, which is based on the IEEE 802.15.4 standard, operates at low data rates of up to 250 kbps with low power consumption, and supports mesh network topologies where nodes can route traffic for each other, thereby extending the effective coverage area. This mesh capability is a noteworthy advantage, as it allows the network to cover a larger area than any single node could reach on its own, making ZigBee suitable for building automation and industrial monitoring applications. Farahani [2011] provides a comprehensive overview of ZigBee networks, covering the protocol stack and practical deployment considerations.

All three of these short-range technologies are valuable for their respective use cases, but they share a key limitation when it comes to campus-scale deployments: their transmission range is generally less than one hundred metres, which means that covering an entire university campus would require a dense infrastructure of access points or repeaters, significantly increasing both initial installation costs and ongoing maintenance requirements.

2.2.3 Low-Power Wide-Area Network Technologies (LPWAN)

LPWAN technologies were developed specifically to address the limitations of short-range protocols by trading data rate for range and power efficiency, thereby providing connectivity over distances of multiple kilometres while allowing battery-operated devices to last for years without replacement. This fundamental trade-off makes LPWAN solutions ideal for applications such as smart campuses, smart cities, and agricultural monitoring, where sensors need to be spread over wide areas and access to mains power is not always available [Raza et al., 2017].

The three most prominent LPWAN technologies are LoRaWAN, Sigfox, and Narrowband Internet of Things (NB-IoT), each of which has its own set of characteristics, deployment models, and trade-offs that must be considered when selecting a technology for a particular use case.

LoRaWAN

LoRaWAN, which stands for Long Range Wide Area Network, is an open standard built on top of the Long Range (LoRa) physical layer modulation developed by Semtech. It operates in unlicensed Industrial, Scientific and Medical (ISM) frequency bands, specifically 868 MHz in Europe, 915 MHz in North America, and 433 MHz in parts of Asia [Semtech Corporation, 2021].

From a technical standpoint, LoRaWAN is capable of delivering communication

ranges of 2 to 15 km in rural areas and approximately 1 to 5 km in urban environments where obstacles are present. Data rates vary between 0.3 and 50 kbps depending on the spreading factor that is configured, and power consumption is extremely low, enabling battery lifetimes of 5 to 10 years for typical sensor nodes. The network employs a star-of-stars topology in which gateways forward received packets to a central network server, and a single gateway can handle communication with tens of thousands of devices [Augustin et al., 2016].

One of the key strengths of LoRaWAN is its adaptive data rate mechanism, which allows the network to dynamically adjust spreading factors and transmission power based on link quality, thereby optimizing both energy consumption and network capacity. The protocol defines three device classes, designated A, B, and C, each offering different trade-offs between downlink latency and power consumption, with Class A being the most energy-efficient option as devices only open short receive windows after transmitting, which is the configuration typically used for sensor nodes [LoRa Alliance, 2020].

Security in LoRaWAN is implemented through AES-128 encryption at both the network and application layers, providing end-to-end data protection. Device authentication can be performed either through Over-The-Air Activation (OTAA), where session keys are derived dynamically during the join procedure, or through Activation By Personalization (ABP), where keys are pre-provisioned on the device [Haxhibeqiri et al., 2018].

What is especially important for the purposes of this dissertation is that LoRaWAN allows organizations to deploy and manage their own network infrastructure. Unlike technologies that depend on a telecom operator, a university can purchase and install its own gateways, run its own network server, and maintain full control over the data flow. This independence has significant practical implications, including data sovereignty, the absence of recurring subscription costs, and no dependency on third-party service availability.

Sigfox

Sigfox is a proprietary LPWAN technology that operates as a network service provider, managing the infrastructure and connectivity on behalf of its customers. Unlike LoRaWAN, Sigfox does not allow organizations to deploy their own infrastructure, as all communication goes through the Sigfox-operated public network [Mekki et al., 2019].

While Sigfox achieves impressive communication ranges of 10 to 50 km in rural areas, this comes at the cost of extremely limited data rates of approximately 100 bps, along with severe message restrictions that limit devices to a maximum of 140 uplink and 4 downlink messages per day, with uplink payloads limited to just 12 bytes. These constraints effectively make Sigfox suitable only for very simple use cases such as basic asset tracking or alarm notifications. For a smart campus deployment with

potentially hundreds of sensors reporting environmental data at regular intervals, these limitations are far too restrictive.

Additionally, the lack of private network deployment means that the institution has no control over the infrastructure and becomes entirely dependent on Sigfox network availability in the area, which creates both a vendor lock-in risk and a practical coverage risk.

NB-IoT (Narrowband IoT)

NB-IoT is a cellular LPWAN technology that was standardized by 3GPP as part of LTE Release 13 and operates in licensed spectrum using existing cellular infrastructure, which brings advantages in terms of reliability and quality of service [3GPP, 2016].

NB-IoT offers data rates of up to approximately 200 kbps, which is significantly higher than both LoRaWAN and Sigfox. Coverage depends on the existing cellular infrastructure and typically ranges from 1 to 10 km, while power consumption is lower compared to standard LTE but somewhat higher than LoRaWAN for equivalent data transmission tasks. The technology also delivers good indoor penetration and benefits from the global reach of cellular networks [Wang and Chen, 2017].

However, NB-IoT shares the same key limitation as Sigfox for the purposes of this work: it cannot be deployed as a private network. Each device requires a SIM card and a subscription with a cellular operator, resulting in recurring per-device costs that accumulate as the deployment scales up. For an academic institution operating with limited budgets and seeking full control over its infrastructure, this cost structure and the associated dependency on external operators represent serious drawbacks.

2.2.4 Why LoRaWAN? A Comparison

Table 2.1 presents a comparison of the key characteristics of the three LPWAN technologies discussed in the previous sections.

Table 2.1: LPWAN technologies compared.

Criterion	LoRaWAN	Sigfox	NB-IoT
Range	2–15 km	10–50 km	1–10 km
Data Rate	0.3–50 kbps	∼100 bps	∼200 kbps
Battery Life	5–10 years	10+ years	2–5 years
Deployment	Private or public	Public only	Public only
Spectrum	Unlicensed (ISM)	Unlicensed (ISM)	Licensed (cellular)
Infrastructure Control	Full	None	None
Initial Cost	Moderate	Low	Low
Recurring Cost	None (private)	Subscription	Subscription
Indoor Penetration	Excellent	Excellent	Excellent

For a smart campus implementation, LoRaWAN emerges as the most suitable choice among the three alternatives, for several reasons that are worth examining in detail.

In terms of infrastructure control and data sovereignty, an academic institution derives significant benefits from deploying and managing its own network, as it gains complete control over configuration, capacity allocation, and data handling, which is particularly valuable for a research university that may want to experiment with network parameters or integrate custom applications [Haxhibeqiri et al., 2018].

Regarding economic sustainability, once the initial investment in gateways has been made, a private LoRaWAN network does not incur recurring fees regardless of how many devices are added, which is a cost model that aligns well with academic budgets and allows the deployment to grow incrementally without proportional cost increases.

The data rates offered by LoRaWAN, reaching up to 50 kbps, are more than sufficient for campus monitoring applications that typically involve periodic sensor readings transmitted at intervals ranging from minutes to hours. Higher data rates offered by cellular technologies would provide capability that is not actually needed for these use cases, while consuming more power in the process [Mekki et al., 2019].

In terms of coverage, a small number of strategically placed gateways, typically three to five for a university campus, can provide comprehensive coverage of both outdoor areas and indoor spaces, which keeps the infrastructure requirements significantly simpler compared to short-range alternatives that would require dozens or hundreds of access points. Energy efficiency is another important consideration, as battery-powered sensors can operate for years without maintenance, enabling deployment in locations that lack power infrastructure such as rooftops, outdoor areas, and remote corners of buildings [Augustin et al., 2016].

The open and vendor-neutral nature of the LoRaWAN standard ensures a diverse marketplace of hardware vendors, software vendors, and network server implementations, which prevents vendor lock-in and provides flexibility in choosing components [LoRa Alliance, 2020]. Furthermore, operating a LoRaWAN network provides

students in computer science and related fields with hands-on learning opportunities, which aligns with the educational mission of the university.

Multi-year deployment studies have validated these advantages in practice. Mikhaylov et al. [2021] documented the operation of a large-scale indoor LoRaWAN network comprising over 300 devices for more than two years, reporting packet delivery ratios exceeding 90% for properly configured networks while also identifying practical challenges related to spreading factor optimization and seasonal effects on radio propagation. Their findings provide valuable benchmarks for planning similar deployments.

2.3 Environmental and Campus Monitoring

2.3.1 Types of Sensors for Environmental Monitoring

Environmental monitoring in smart campus deployments relies on a variety of sensor types, each of which measures specific physical or chemical parameters that are relevant to campus operations, occupant comfort, and sustainability goals.

Temperature and humidity sensors provide essential data for HVAC control, comfort assessment, and building performance evaluation. In the context of LoRaWAN deployments, many commercially available sensors combine temperature and humidity measurement in a single compact device with multi-year battery life.

Air quality sensors measure parameters such as CO_2 concentration, Volatile Organic Compound (VOC) levels, and particulate matter including PM2.5 and PM10, which are important for assessing indoor environmental quality in spaces such as classrooms and offices. Elevated CO_2 concentrations have been linked to reduced cognitive performance, making this type of monitoring particularly relevant in educational settings. Alvear-Puertas et al. [2022] provide an overview of IoT-based air quality monitoring approaches and discuss the challenges involved in deploying such systems in urban environments.

Occupancy and motion sensors, which employ technologies such as passive infrared detection, ultrasonic sensing, and people-counting algorithms, are used to determine whether spaces are occupied and sometimes to estimate the number of occupants. This data supports energy-saving strategies by enabling demand-driven climate control rather than fixed-schedule operation.

Energy monitoring sensors measure electrical consumption at various levels, from whole-building meters down to individual circuit monitors. When combined with occupancy data, energy monitoring helps identify waste and optimize consumption patterns across the campus.

Weather stations collect outdoor meteorological data including temperature, humidity, wind speed and direction, atmospheric pressure, and rainfall, supporting both

operational decisions and environmental research activities.

The diversity of sensor types that may be deployed in a smart campus underscores the need for a flexible and extensible data pipeline, as the system architecture must be able to accommodate new sensor types without requiring significant changes to the core infrastructure.

2.3.2 IoT Data Management and Visualization Platforms

A complete IoT platform for campus monitoring involves several interconnected components that together handle the full data journey from acquisition through processing to presentation.

Components of an IoT Platform

Device management encompasses the registration, configuration, and ongoing monitoring of sensors and gateways, including maintaining an inventory of deployed devices, tracking their operational status, and managing firmware updates when necessary.

Data ingestion is responsible for receiving data from devices and routing it into the processing pipeline. In LoRaWAN deployments, the network server handles the initial reception of data and makes it available through integration points, most commonly via MQTT topics.

Message brokering provides the asynchronous communication backbone between the various system components, with message brokers serving to decouple data producers from consumers and enable independent scaling and fault isolation. The choice of message broker, whether it be Apache Kafka, RabbitMQ, or another system, has significant implications for system complexity, performance, and operational requirements [Dobbelaere and Esmaili, 2017].

Data storage for IoT applications needs to be optimized for time-series sensor data, as traditional relational databases often fall short when dealing with the write-heavy, time-ordered workloads that IoT systems generate. Time-series databases and columnar stores are better suited to provide the performance required for both real-time ingestion and analytical queries [Naqvi et al., 2017].

Data visualization and analytics transform the stored data into actionable insights through dashboards, reports, alerts, and ad-hoc queries. The presentation layer needs to support both real-time views of current conditions and historical views that enable trend analysis over longer time periods.

LoRaWAN Network Servers

LoRaWAN network servers are critical components in any LoRaWAN deployment, as they manage the communication between gateways and application servers, handle device authentication, perform packet deduplication when multiple gateways receive the same transmission, control adaptive data rates, and decrypt payloads before passing data to downstream applications.

ChirpStack, formerly known as LoRa Server, is the most widely deployed open-source LoRaWAN network server, developed by Orne Brocaar and maintained by an active community. It provides a comprehensive set of features suitable for both experimental and production deployments [Brocaar, 2023], including multi-tenancy support with role-based access control, a web interface and APIs for device management, native integrations with MQTT and other platforms, and support for all LoRaWAN regional parameters including the EU868 frequency band used in Portugal.

The architecture of ChirpStack comprises several components: the Gateway Bridge, which communicates with physical gateways; the Network Server, which handles the LoRaWAN protocol; and the Application Server, which manages device registrations and data integrations. For the purposes of this dissertation, ChirpStack's MQTT integration is of particular importance, as it serves as the data source from which the MQTT bridge components consume device data.

Other open-source alternatives exist, notably The Things Network and its self-hosted variant The Things Stack. ChirpStack was selected for this work due to its maturity, its straightforward self-hosted deployment model, and its flexible integration options.

Communication Protocols for IoT Platforms

Within an IoT platform, choosing a messaging protocol for inter-component communication is a critical architectural decision, and two protocols stand out as the most commonly used: MQTT and AMQP.

MQTT is a lightweight publish-subscribe messaging protocol that was originally developed by IBM for the purpose of monitoring oil pipelines over satellite links. Its design prioritizes low bandwidth consumption and reliability over unreliable networks, characteristics that have made it the de facto standard for IoT device communication [OASIS Standard, 2019]. MQTT uses a topic-based publish-subscribe model in which publishers send messages to named topics, subscribers receive messages from topics they are interested in, and a broker intermediates all communication, thereby decoupling publishers from subscribers. The protocol supports three quality of service levels: QoS 0 for at-most-once delivery, QoS 1 for at-least-once delivery, and QoS 2 for exactly-once delivery, providing flexibility in the trade-off between reliability and overhead [Naik, 2017].

AMQP, the Advanced Message Queuing Protocol, is more feature-rich and is commonly used in enterprise systems. It offers sophisticated routing capabilities including direct exchanges, topic exchanges, and fan-out patterns. RabbitMQ, one of the most popular message brokers, implements AMQP as its primary protocol and provides features such as message acknowledgement, dead-letter queues, and priority queues, all of which are useful for building reliable data pipelines [Videla and Williams, 2012].

Integrating MQTT with a message broker such as Kafka or RabbitMQ is a common architectural pattern in IoT systems, where MQTT handles the device-facing communication efficiently while the message broker provides the durability, routing capabilities, and scalability needed for the backend data pipeline. This pattern is central to the architecture proposed in this dissertation and is discussed in detail in Chapter 3.

Dobbelaere and Esmaili [2017] present a detailed comparison of Kafka and RabbitMQ, noting that Kafka excels at high-throughput log processing and event streaming with its persistent, partition-based architecture, whereas RabbitMQ is often more suitable for scenarios that require flexible routing, lower operational complexity, and lower latency at moderate throughput levels. This comparison was influential in the architectural evolution documented in this dissertation, as the initial use of Kafka was subsequently reconsidered in favour of RabbitMQ.

2.3.3 Time-Series Databases

The choice of database for storing and querying IoT sensor data is a critical architectural decision, as the database must handle continuous write-heavy workloads while also supporting efficient analytical queries over potentially large time ranges. Three databases are commonly considered for time-series workloads in IoT contexts: ClickHouse, TimescaleDB, and InfluxDB. Each has distinct architectural characteristics that make it more or less suitable depending on the specific requirements of the deployment.

ClickHouse is a columnar analytical database developed by Yandex that is optimised for online analytical processing queries over large datasets. Its column-oriented storage engine achieves high compression ratios, particularly with repetitive sensor data, and it executes aggregation queries with exceptional speed due to its vectorised query execution engine [ClickHouse Inc., 2023].

TimescaleDB is an extension of PostgreSQL that adds time-series capabilities on top of a mature relational database. It benefits from full SQL compatibility and the rich ecosystem of PostgreSQL tools, which makes it attractive for teams that already have PostgreSQL expertise. However, the row-oriented storage model inherited from PostgreSQL results in significantly higher disk usage compared to columnar alternatives [Naqvi et al., 2017].

InfluxDB is a purpose-built time-series database that uses a custom storage engine optimised for time-stamped data. It provides a specialised query language and achieves good compression for time-series workloads, though its query flexibility is more limited compared to SQL-based alternatives.

Barez et al. [2023] conducted a systematic benchmarking study comparing these databases, and their findings are particularly relevant to the technology selection made in this dissertation. Table 2.2 summarises the key characteristics of the three databases based on their findings and other benchmark evaluations reported in the literature.

Table 2.2: Comparison of time-series databases for IoT workloads.

Criterion	**ClickHouse**	**TimescaleDB**	**InfluxDB**
Storage model	Columnar	Row-based (PostgreSQL)	Custom TSM
Ingestion rate	∼4M metrics/s	∼1.3M metrics/s	∼1.3M metrics/s
Disk usage (relative)	Low	Very high (∼20×)	Lowest
Simple query perf.	Fast	Moderate	Fast
Complex query perf.	Fastest	Moderate	Moderate
SQL support	Full	Full (PostgreSQL)	InfluxQL / Flux
Compression ratio	Excellent	Moderate	Good
Horizontal scaling	Native sharding	Limited	Enterprise only
Licence	Apache 2.0	Apache 2.0	MIT (OSS) / Proprietary

The benchmarking results from Barez et al. [2023] demonstrated that ClickHouse achieves approximately three times higher ingestion throughput compared to both TimescaleDB and InfluxDB, and that for complex analytical queries involving aggregations, groupings, and filtering over large time ranges, ClickHouse significantly outperforms both alternatives. TimescaleDB benefits from its PostgreSQL foundation in terms of SQL compatibility and ecosystem tooling, but its row-based storage model results in disk usage that can be up to twenty times higher than ClickHouse for equivalent datasets.

For the smart campus monitoring system described in this dissertation, ClickHouse was selected on the basis of several factors: its superior ingestion performance is well suited to handle continuous streams of sensor data from potentially hundreds of devices, its columnar storage achieves excellent compression with the repetitive nature of environmental readings, its fast aggregation query performance supports responsive dashboard visualisations, and its open-source licence with native horizontal scaling capabilities aligns with the project's commitment to open technologies and future growth. The main trade-off is that ClickHouse lacks the full relational capabilities of TimescaleDB, but since the application's metadata is stored separately in PostgreSQL, this limitation does not affect the system's functionality.

Chapter 3

Proposed Architecture

When I set out to build the smart campus monitoring platform, I was confronted with a number of significant architectural decisions that needed to be made relatively early in the process, the kind of decisions that, once committed to, become deeply embedded in everything that follows. This chapter presents the architecture I designed and, perhaps more importantly, documents how it evolved as I moved from theoretical design to actual implementation and began to understand what works well in practice versus what appears adequate only on paper. I started with a first version of the architecture based on Apache Kafka as the central message streaming platform, which seemed like a reasonable choice at the time given its reputation for scalability. However, as development progressed, I ended up conceiving a second version of the architecture that replaces Kafka with RabbitMQ and introduces a significantly enhanced Rust middleware layer. Both versions are documented in this chapter because the journey between them was itself a valuable learning experience that led to the final design. The remainder of this chapter describes the design principles that guided both versions, then present each architecture in detail, and conclude with a comparison that explains why I made the transition.

3.1 Design Principles

Before choosing specific technologies, I established core design principles. These principles were meant to guide all architectural decisions, regardless of the tools or platforms used. These principles remained the same throughout the development from version 1 to version 2.

The foundational principle is the separation of concerns, which dictates that each system component should be assigned a singular, clearly delineated responsibility. Device connectivity is managed by one component, protocol translation by another, message routing by a third, and data storage by a fourth. Furthermore, the interfaces that connect these components are explicitly defined, thereby ensuring that changes to one component do not affect the others unexpectedly.

The second principle is the adoption of a microservices architecture over a monolithic design. I decomposed the application layer into independent services that communicate with each other but do not depend on each other for their continued operation. Each service can be developed, tested, deployed, and scaled independently, which is not merely an architectural preference but a practical necessity for a system that is expected to evolve as new requirements emerge over time [Newman, 2015].

The third principle is event-driven communication, whereby components exchange messages asynchronously rather than making synchronous calls to each other. This approach provides resilience, since a temporary failure in one component does not immediately break everything downstream, and it also means that data producers and consumers do not need to be tightly coupled in time, as a producer can emit data whenever it has data available and a consumer can process it whenever it is ready [Hohpe and Woolf, 2004].

The fourth principle is scalability by design. Having this in mind from the very beginning, I ensured that the architecture could scale horizontally at multiple levels, whether that means adding more gateways for increased coverage, deploying additional bridge instances for greater device capacity, or spinning up more microservice replicas to handle application load. Scalability was treated as a first-class requirement rather than something to be addressed as an afterthought.

The fifth principle is a commitment to open technologies. I selected open-source software and open standards throughout the system in order to avoid vendor lock-in, to maintain full visibility into how the components work, and to align with how a University environment should operate. This choice also has educational value, as students can study, modify, and learn from the actual system rather than being constrained by proprietary tools.

3.2 Justification for Rust over Go

A crucial technological determination within this architectural framework involved the selection of a programming language for the elements of the MQTT bridge and middleware. Considering the pivotal role of this component in the data pipeline, through which all sensor data are traversed, the chosen language significantly influences system performance, resource utilization, and sustained dependability. The two principal options under consideration were Rust and Go, both contemporary system programming languages characterized by robust concurrency capabilities and increasing prevalence in infrastructure software development.

Rust was ultimately selected over Go for several reasons that are particularly relevant to the requirements of a IoT protocol bridge. The most significant factor is Rust's memory management model, which uses an ownership and borrowing system enforced at compile time rather than relying on a garbage collector at runtime. This design eliminates the possibility of garbage collection pauses, which in Go can

introduce latency spikes of several milliseconds even with the improvements made in recent Go versions [Klabnik and Nichols, 2023]. For a bridge component that must process messages with consistently low latency, this predictability is a critical advantage.

Table 3.1 presents a comparison of key performance characteristics between Rust and Go based on publicly available benchmark data from the programming language benchmarks community and independent evaluations [Costanzo et al., 2021].

Table 3.1: Performance comparison between Rust and Go for systems programming tasks. [Costanzo et al., 2021]

Metric	**Rust**	**Go**
HTTP throughput (requests/s)	∼60,000	∼40,000
Memory usage (idle service)	1–3 MB	8–15 MB
Garbage collection pauses	None	0.5–5 ms
Binary size (static, stripped)	2–5 MB	8–15 MB
CPU-bound task performance	Baseline	1.3–2× slower
Concurrency model	async/await (Tokio)	goroutines
Memory safety	Compile-time (ownership)	Runtime (GC)

Beyond raw performance, Rust offers additional advantages that are relevant to this project. The type system and ownership model catch entire categories of bugs at compile time, including data races in concurrent code, which is particularly important for a multi-threaded bridge component that handles shared state across concurrent MQTT connections [Jung et al., 2021]. The resulting binaries are also considerably smaller and have no runtime dependencies, which simplifies containerised deployment and reduces the attack surface.

It is important to recognize that Rust's more challenging learning curve and its comparatively protracted compilation times present a pragmatic compromise. Go's inherent simplicity and accelerated development cycle render it a highly suitable option for numerous backend services; indeed, the application-layer microservices within this system could have been effectively constructed using Go. Nevertheless, considering the particular demands of the protocol bridge specifically, the need for sub-millisecond latency, a minimal memory footprint, predictable performance under heavy load, and compile-time safety assurances Rust emerged as the more fitting selection. The performance characteristics observed during testing, as documented in Chapter 5, confirm that this decision was justified.

3.3 Architecture Version 1: Kafka-Based Design

The first version of the architecture was designed with a strong focus on scalability and enterprise-grade event streaming capabilities.

3.3.1 Overview

In version 1, the data flows through several distinct layers:

1. **Perception layer:** IoT devices equipped with LoRaWAN radios transmit sensor readings over the air.
2. **Network layer:** LoRaWAN gateways receive these transmissions and forward them to ChirpStack, which serves as the LoRaWAN network server.
3. **Protocol bridge layer:** Multiple MQTT bridge instances written in Rust subscribe to ChirpStack's MQTT topics, receive the data, apply the necessary transformations, and republish everything to Apache Kafka.
4. **Message streaming layer:** Apache Kafka serves as the central event streaming platform, providing persistent storage of all sensor data and making it available to whatever downstream services need to consume it.
5. **Data storage layer:** ClickHouse consumes data from Kafka and provides high-performance analytical storage that is optimized for time-series queries.
6. **Application layer:** A set of specialized microservices handles asset management, topic management, user management, and the frontend presentation.

3.3.2 Component Details

ChirpStack

ChirpStack serves as the LoRaWAN network server at the centre of device communication, managing the entire LoRaWAN protocol including device authentication, packet deduplication, adaptive data rate control, and payload decryption. When ChirpStack finishes processing a device uplink, it publishes the decoded data to an MQTT broker following a predictable topic structure that is based on the application and device identifiers. For example, an uplink event from a specific device is published to a topic with the following structure:

Listing 3.1: ChirpStack MQTT topic structure

```
application/{application_id}/device/{dev_eui}/event/up
```

This MQTT output is the point at which the rest of the system begins consuming device data.

MQTT Bridge Instances

The MQTT bridge is a component that I built specifically for this architecture, written in Rust using the Axum web framework. It serves as the critical link that connects ChirpStack's MQTT output to the message streaming layer.

In version 1, I deployed three bridge instances running in parallel, with each instance handling a subset of devices. The instances registered themselves with a Topics Manager service, which was responsible for assigning device topics to each instance using a distribution algorithm. This design enables horizontal scaling, since when the number of devices increases, additional bridge instances can be deployed and the Topics Manager rebalances the load across them.

Each bridge instance performed the following operations: it subscribed to the assigned ChirpStack MQTT topics, parsed the incoming JSON payloads in order to extract the DevEUI, deduplication ID, and raw sensor data, transformed the raw measurements into a structured format with labelled fields, and published the transformed data to Kafka topics with one topic allocated per device.

Apache Kafka

Apache Kafka served as the backbone of the version 1 data pipeline, providing distributed, fault-tolerant, and persistent message storage with high throughput characteristics. Kafka's topic-partition model allows for parallel consumption, and its configurable retention policies make it possible to keep historical data available for replay if needed [Kreps et al., 2011].

The Topics Manager microservice was responsible for dynamically creating and configuring Kafka topics as new devices were registered in the system. It also handled the assignment of topics to bridge instances and maintained the mapping between topics and instances in ClickHouse.

ClickHouse

ClickHouse serves as the analytical database layer and is optimized for time-series workloads, which is precisely the type of data that an IoT monitoring system generates. Its columnar storage engine achieves excellent compression ratios with sensor data, since the readings tend to be repetitive in nature, and it enables fast aggregation queries even over very large datasets. Data flows into ClickHouse either through Kafka engine tables that consume directly from Kafka topics or through dedicated ingestion services [ClickHouse Inc., 2023].

Application Microservices

In version 1, the application layer comprised the following services. The Assets Manager handles device and gateway metadata, including registration, configuration, location information, and the association of devices with specific applications or tenants. It communicates with ChirpStack's APIs to keep device information synchronized and uses ClickHouse to store metadata. The Topics Manager manages the lifecycle of Kafka topics, creating new ones when devices are registered, assign-

ing them to bridge instances, and monitoring their health. This service is central to the distributed bridge architecture because it determines which bridge instance is responsible for which devices. The Users Manager takes care of authentication, authorization, and user management, implementing role-based access control with user credentials and tokens stored in PostgreSQL. The Frontend is a web application built with Next.js that provides dashboards for viewing sensor data, interfaces for managing devices and gateways, alert configuration, and system administration capabilities.

3.4 Architecture Version 2: RabbitMQ with Rust Middleware

3.4.1 Motivation for the Revision

Although the Kafka-based architecture was technically sound, with a design that could handle the expected data volumes, practical issues arose during development. These issues led me to reconsider the choice of message broker.

Operational complexity was the initial hurdle. Apache Kafka is a powerful platform, but it introduces considerable operational overhead. Managing a Kafka cluster involves a few key tasks. You'll need to deal with ZooKeeper or the KRaft controller, depending on your setup. You'll also be configuring replication factors, keeping an eye on how partitions are distributed, and figuring out what to do when a broker goes down. For a campus deployment that is likely to be maintained by a small team, potentially including students, this level of operational complexity seemed disproportionate to the actual scale of data being processed [Narkhede et al., 2017].

The second issue was resource consumption. Kafka is designed for massive-scale streaming workloads, and its resource profile reflects this design philosophy. Even a minimal cluster consumes significant amounts of memory and CPU resources. Given that the system was expected to handle hundreds of messages per second rather than millions, the resources that Kafka required appeared to be out of proportion with the actual needs.

The third issue related to routing requirements. The version 1 design used one Kafka topic per device, which meant that as the number of devices grew, so did the number of topics. While Kafka can technically handle this, the management overhead was unnecessary for this use case, since what I actually needed, namely distributing messages from devices to a set of consuming services, could be accomplished more naturally with the exchange and queue model that RabbitMQ offers.

The fourth issue was latency characteristics. RabbitMQ is designed for low-latency message delivery, whereas Kafka is optimized for high-throughput batched processing. In a campus monitoring system where real-time alerting is an important requirement, RabbitMQ's delivery characteristics are a better fit [Dobbelaere and Esmaili,

2017].

The fifth issue concerned the overall infrastructure footprint. RabbitMQ runs as a single Erlang application with built-in clustering support, eliminating the need for separate coordination services such as ZooKeeper, which simplifies deployment, monitoring, and maintenance, all of which matter when resources are limited.

3.4.2 Revised Architecture Overview

Version 2 retains the fundamental structure and design principles from version 1 but replaces Kafka with RabbitMQ as the message broker and enhances the MQTT bridge with additional Rust middleware for data transformation and routing.

The data flow in version 2 proceeds as follows:

1. IoT devices transmit sensor readings over LoRaWAN.
2. Gateways forward the received packets to ChirpStack.
3. ChirpStack publishes the decoded data to its MQTT broker.
4. Rust-based MQTT bridge instances consume data from ChirpStack's MQTT broker, apply the necessary transformations, and publish the results to RabbitMQ exchanges.
5. RabbitMQ routes messages to the appropriate queues based on exchange bindings and routing keys.
6. ClickHouse ingestion services pull data from RabbitMQ queues and write it to the analytical database.
7. Application microservices consume from their relevant queues to perform their respective functions.
8. The frontend queries ClickHouse for historical data and subscribes to RabbitMQ for live updates.

3.4.3 Key Differences from Version 1

The most significant change is the replacement of the message broker. RabbitMQ replaces Kafka as the central message router, and instead of Kafka's topic-partition model, the system now uses RabbitMQ's exchange-queue model with topic exchanges that enable flexible routing based on routing keys. This eliminates the one-topic-per-device approach and results in a considerably cleaner messaging topology.

The Rust middleware in version 2 has been enhanced beyond simple protocol translation. The MQTT bridge now includes data validation, enrichment with metadata such as timestamps and signal quality indicators, and configurable routing

logic. Rust was chosen for this component because of its performance characteristics, memory safety without garbage collection overhead, and strong concurrency support, all of which are essential qualities for a component that sits on the critical data path [Jung et al., 2021; Klabnik and Nichols, 2023].

Topic management has also been simplified. With RabbitMQ, the Topics Manager service no longer needs to create and manage potentially hundreds of individual topics. Instead, it configures a smaller set of exchanges and queue bindings, with routing handled through routing keys rather than through topic proliferation.

The overall resource requirements have been reduced as well. RabbitMQ requires less memory and CPU than a Kafka cluster for the message volumes I expect, and the elimination of ZooKeeper removes an entire component from the deployment.

3.5 Data Flow Architecture

Regardless of whether Kafka or RabbitMQ is used as the message broker, the end-to-end data flow through the system follows the same general pattern:

1. **Sensor transmission:** An IoT device encodes its reading into a compact payload and transmits it as a LoRaWAN packet.
2. **Gateway reception:** One or more gateways within range receive the transmission and forward the raw packet to ChirpStack over IP.
3. **Network server processing:** ChirpStack authenticates the device, performs deduplication if multiple gateways received the same transmission, decrypts the payload, and publishes the result to MQTT.
4. **Protocol bridging:** The Rust MQTT bridge subscribes to the relevant device topics, parses the ChirpStack JSON payload in order to extract the DevEUI, deduplication ID, and sensor values, transforms the data into the required structure, and publishes it to the message broker.
5. **Message routing:** The message broker, whether Kafka in version 1 or RabbitMQ in version 2, persists the message and makes it available to downstream consuming services.
6. **Storage ingestion:** ClickHouse receives the data and stores it in optimized columnar tables that are partitioned by time.
7. **Application consumption:** The various microservices consume their relevant data streams in order to perform functions such as alerting, aggregation, and device management.
8. **User access:** The frontend queries ClickHouse for historical data and subscribes to real-time message streams for live dashboard updates.

3.6 Security Considerations

Security has been considered at every layer of the architecture rather than being treated as an addition after the fact.

At the device level, security is provided by the LoRaWAN protocol itself, which implements AES-128 encryption at both the network and application layers. This prevents eavesdropping on transmitted data and ensures that only authenticated devices can join the network. I use OTAA for device activation, which offers stronger security guarantees than the alternative static key approach provided by ABP, since session keys are derived dynamically during the join procedure.

At the network level, all communication between ChirpStack, the MQTT broker, and the message broker is secured using TLS encryption. Internal services communicate over a private network segment, and firewall rules restrict access to only the necessary ports.

Application-level security hinges on the Users Manager service, which governs access via roles. Token-based authentication safeguards every API endpoint, and all access attempts are logged, creating an audit trail.

Data stored within ClickHouse is secured on encrypted volumes. Furthermore, access to the database itself is strictly limited to services that have been authorized.

3.7 How the Two Versions Compare

Table 3.2 summarizes the key differences between the two architectural versions.

Table 3.2: How the two versions differ.

Aspect	**Version 1 (Kafka)**	**Version 2 (RabbitMQ)**
Message broker	Apache Kafka	RabbitMQ
Routing model	Topic per device	Exchange + routing keys
Persistence	Log-based (Kafka)	Queue-based + ClickHouse
Operational complexity	High	Moderate
Resource requirements	High	Moderate
Latency	Higher (batched)	Lower (per-message)
Scalability ceiling	Very high	High
Middleware	Rust MQTT bridge	Enhanced Rust middleware
Dependencies	ZooKeeper/KRaft	Erlang runtime

The trade-off between the two versions can be summarised as follows: version 1 offers a higher theoretical scalability ceiling due to Kafka's distributed log architecture, while version 2 provides a more pragmatic solution that is easier to deploy, operate, and maintain at the scale typical of a campus deployment. This said, the decision to move to version 2 was driven by the recognition that architectural choices should

be guided by actual requirements and constraints rather than by what is theoretically possible. Both iterations retain the essential design tenets and key elements: ChirpStack, protocol bridges built with Rust, ClickHouse, and a microservices application layer. The transition was specifically about selecting the most appropriate message broker for the context in which the system would operate, which is a decision that many IoT practitioners are likely to face as they move from prototyping to production deployments.

Chapter 4

Implementation

This chapter describes how the smart campus monitoring system was actually built, covering the main software components that were developed, the configuration of the infrastructure elements, and the approach taken for deployment. The chapter follows the data as it flows through the system, beginning with the configuration of the LoRaWAN network server and ending with the frontend application that users interact with.

4.1 Development Environment and Tools

The system was built using a combination of programming languages and frameworks, each selected based on the specific requirements of the component it was used for.

Rust was chosen as the implementation language for the MQTT bridge and middleware components. Since the bridge sits directly on the critical data path through which all sensor data flows, I needed a language that could deliver both high performance and strong guarantees regarding memory safety. Rust's ownership model ensures that the compiler catches memory-related bugs at compile time rather than at runtime, and the absence of a garbage collector means that latency remains predictable even under sustained load [Klabnik and Nichols, 2023]. It was used the Tokio runtime for asynchronous operations, which allows the bridge to handle many concurrent network connections efficiently without the overhead of spawning a separate thread for each one. For the HTTP endpoints that the bridge exposes, the selected web framework was Axum, which provided the functionality and simplicity I needed with minimal overhead [Tokio Contributors, 2023].

Next.js was selected for the frontend application because it provides server-side rendering capabilities out of the box along with React components, which together made it considerably easier to build an interactive dashboard [Vercel, 2023].

Docker was used for containerizing every component in the system, including Chirp-

Stack, the custom microservices, the message brokers, ClickHouse, and all supporting services. Containerization ensured that the same environments could be replicated across development, testing, and production without compatibility issues. Docker Compose was used to orchestrate the entire stack during development, while container orchestration tools could be employed for production deployments [Merkel, 2014].

Git was used for version control throughout the project, with the MQTT bridge code hosted on GitHub[1] in order to facilitate collaborative development and change tracking.

Development was carried out on a standard Linux workstation, with all infrastructure services running locally in Docker containers during development. This said, there were occasions where something that worked correctly in the local environment behaved differently in testing, which required careful investigation and debugging.

4.2 ChirpStack Configuration

ChirpStack was implemented using the official Docker images made available by the project. This deployment architecture encompasses three primary elements: the ChirpStack Gateway Bridge, the ChirpStack server and the necessary databases (PostgreSQL for metadata storage and Redis for caching and device session oversight).

4.2.1 Network Configuration

ChirpStack was configured for the EU868 frequency band, which is the LoRaWAN standard applicable to Portugal. The configuration includes eight channels in the 868 MHz band in accordance with the EU868 regional parameters defined by the LoRa Alliance. Adaptive data rate was enabled, which allows ChirpStack to automatically adjust the spreading factor and transmission power for each device based on the observed link quality. For device activation, I opted for OTAA, which allows devices to derive their session keys dynamically during the join procedure rather than relying on static pre-provisioned keys, thereby providing stronger security guarantees.

4.2.2 Multi-Tenancy Setup

Multi-tenancy is a core feature of ChirpStack, and I configured it to reflect the university's organizational structure. At the highest level, Organizations represent the university's different departments and research groups. Within each organization, Applications group devices based on their function or where they are used. For

[1] https://github.com/jcondeco207/mqtt_bridge

example, applications could be for environmental monitoring in a specific building or energy monitoring across the entire campus. Finally, at the most detailed level, individual Devices represent the sensor nodes, each identified by a unique DevEUI and linked to a specific application.

This hierarchical structure allows different groups to manage their own devices and applications independently while sharing the same LoRaWAN infrastructure, which provides a form of organizational isolation that prevents configuration changes in one group from inadvertently affecting another.

4.2.3 MQTT Integration

ChirpStack was configured to publish device events to an MQTT broker running on the same host, using a topic structure that follows ChirpStack's standard format, as shown in the in 4.1:

Listing 4.1: ChirpStack MQTT topic pattern for uplink events

```
application/{application_id}/device/{dev_eui}/event/up
```

This is the integration point at which the MQTT bridge instances connect and begin consuming device data. During development, the `mosquitto_sub` command-line tool proved invaluable for observing the raw message stream and verifying that ChirpStack was publishing data in the expected format 4.2:

Listing 4.2: Subscribing to all device events for an application

```
mosquitto_sub -h 192.168.3.231 \
  -t "application/{app_id}/device/+/event/+" -v
```

This capability to directly observe the messages flowing through the system was particularly useful during debugging, as it allowed me to quickly determine whether issues were originating at the ChirpStack level, at the MQTT broker level, or within the bridge itself.

4.3 MQTT Bridge Implementation

The MQTT bridge is the most significant piece of custom software developed as part of this work. It serves as the critical link between ChirpStack's MQTT output and the downstream message broker, meaning that if this component fails, the entire data pipeline is interrupted. Writing it in Rust gave me confidence in how it handles concurrent connections and memory management.

4.3.1 Project Structure

The Rust project is structured as a conventional Cargo workspace, comprising several key modules. The MQTT client module is tasked with establishing a connection to ChirpStack's broker, overseeing topic subscriptions, implementing reconnection protocols in the event of disconnections, and processing incoming messages. The transformation module is responsible for converting the incoming JSON data from ChirpStack into the format required by the system. The publisher module manages the connection to the downstream message broker, whether Kafka in version 1 or RabbitMQ in version 2, and publishes the transformed messages. Furthermore, the HTTP API module utilizes Axum to provide REST endpoints for bridge registration, health checks, and notifications regarding topic assignments from the Topics Manager.

4.3.2 Bridge Initialization and Registration

When a bridge instance starts up, it goes through a defined sequence of initialization steps. It first generates or loads a unique identifier, then registers itself with the Topics Manager service by sending its identifier, host address, and port number. The Topics Manager acknowledges the registration and responds with a set of device topics that the bridge instance should handle. The bridge then subscribes to those MQTT topics on ChirpStack's broker and begins processing messages from its assigned devices.

This registration-based approach enables workload distribution across multiple bridge instances running in parallel, with each instance handling only its assigned subset of devices. If one instance fails, the Topics Manager can detect the failure and reassign the affected device topics to other running instances, providing a basic level of fault tolerance without introducing excessive complexity.

4.3.3 Data Transformation

From each ChirpStack uplink event, the bridge extracts several key fields. The DevEUI, as defined in the chirpstack documentation, serves as the unique identifier of the device that sent the data. The deduplication ID is a unique identifier for that specific message, which is used to avoid processing the same uplink more than once in cases where multiple gateways forward the same transmission. The application ID indicates which ChirpStack application the device belongs to. The timestamp records when the uplink was received. The sensor data contains the decoded payload with the actual measurements, as ChirpStack applies device-specific decoders that convert the raw LoRaWAN bytes into named fields with appropriate units. Additionally, the RF metadata including Received Signal Strength Indicator (RSSI) and Signal-to-Noise Ratio (SNR) values from the receiving gateway is extracted, as this signal quality information is useful for monitoring network health.

All of this information is structured into a JSON object with a consistent schema regardless of the sensor type, which means that downstream services do not need to implement different parsing logic for different kinds of sensors.

4.3.4 Concurrency Model

The bridge's concurrency model is built on Rust's async/await syntax powered by the Tokio runtime. Each bridge instance runs an asynchronous MQTT client connection to ChirpStack's broker, an asynchronous connection to the message broker for publishing transformed data, and an Axum HTTP server for the management API, all executing concurrently within the same process.

Shared state between these concurrent tasks is managed through Rust's thread-safe primitives including Arc for shared ownership, Mutex for mutual exclusion, and channels for inter-task communication. Rust's ownership model guarantees at compile time that shared state is accessed safely without data races, which is a common source of bugs in concurrent systems written in languages like C or C++ [Balasubramanian et al., 2017].

4.3.5 Performance Considerations

The decision to utilize Rust for the bridge's construction stemmed from a number of performance-oriented considerations. Minimizing latency was a critical factor, given that the bridge's function necessitated a negligible delay between the reception of an MQTT message and its subsequent publication to the downstream broker. Rust's design, characterized by zero-cost abstractions and the absence of a garbage collector, contributes to predictable latency. Furthermore, a low memory footprint was a key requirement, as each bridge instance consumes a relatively small amount of memory, thereby facilitating the operation of multiple instances on hardware with limited resources. High throughput was also a significant consideration, since the asynchronous I/O model allows the bridge to manage numerous concurrent connections without the performance penalties typically associated with thread-per-connection architectures.

4.4 Message Broker Configuration

During the development and experimentation the first solution for this step required the setup and implementation of an Apache Kafka. But as it is explained in this document due to the search for better performance and low resources consumption the second and final version of this experiment replaced Kafka with RabbitMQ.

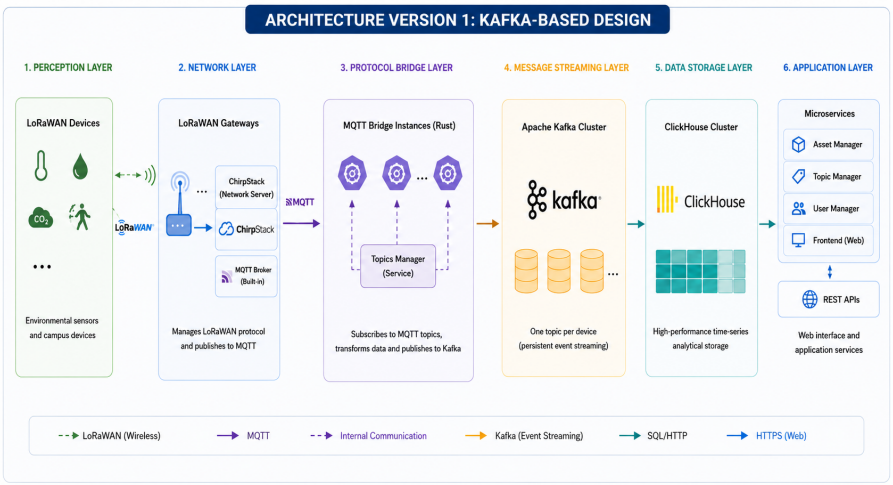

Figure 4.1: Kafka based architeture

4.4.1 Version 1: Apache Kafka

In the first version of the system, Kafka was deployed as a three-broker cluster with ZooKeeper for coordination. The replication factor was set to 3 for data topics, ensuring that the system could tolerate the loss of a single broker without data loss. Raw sensor data was retained for 30 days, with aggregated data subject to longer retention periods. Each device was initially assigned a single partition per topic, with the possibility of increasing partitions for devices that required higher throughput.

The Topics Manager created Kafka topics dynamically as new devices were registered in the system, using the Kafka Admin API and following a naming convention that encoded the application and device identifiers.

4.4.2 Version 2: RabbitMQ

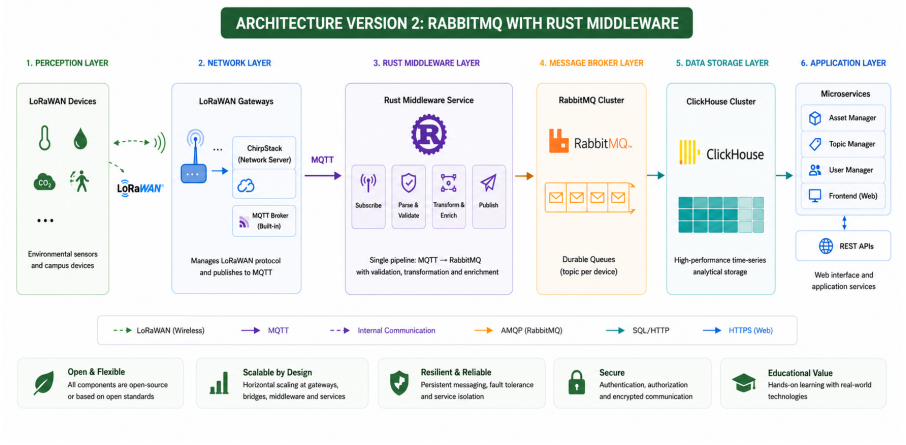

Figure 4.2: RabbitMQ Based Architeture

In the revised version, RabbitMQ replaced Kafka as the message broker, deployed as a single node for development and a three-node cluster for production environments. The messaging topology uses a topic exchange named `sensor_data` for routing device measurements, where messages are published with routing keys that encode the application and device type, such as `environmental.temperature.building-a`. Consumers bind their queues to patterns that match the data they need to receive. A separate direct exchange handles system events such as device registrations, topic assignments, and health checks. All queues are configured as durable with acknowledgement-based delivery, ensuring that messages are not lost if a consumer disconnects temporarily.

This approach is considerably simpler than the Kafka-based version. Instead of managing hundreds of individual topics, the system operates with a small number of exchanges and flexible routing rules. Adding support for new device types or new consuming services requires only the creation of new queue bindings rather than the creation and management of new topics.

4.5 ClickHouse Database Design

In this section it will be explained how Clickhouse data model and logic was implemented.

4.5.1 Schema Design

The ClickHouse schema was designed to handle two distinct types of queries effectively: real-time dashboard queries that retrieve recent data for a specific device or location, and analytical queries that compute trends and aggregations over longer time ranges.

For storing sensor readings, I used the MergeTree engine with monthly partitions 4.3:

Listing 4.3: ClickHouse table definition for sensor readings

```
CREATE TABLE sensor_readings (
    device_id      String,
    application_id String,
    timestamp      DateTime,
    sensor_type    String,
    value          Float64,
    unit           String,
    gateway_id     String,
    rssi           Int16,
    snr            Float32
) ENGINE = ReplacingMergeTree()
PARTITION BY toYYYYMM(timestamp)
ORDER BY (device_id, sensor_type, timestamp);
```

The ordering key `(device_id, sensor_type, timestamp)` was chosen because the most common query pattern involves retrieving recent measurements for a specific device and sensor type. Monthly partitioning simplifies data management, as different retention policies can be applied to older partitions.

4.5.2 Materialized Views for Aggregation

Pre-computed aggregations are maintained through materialized views that automatically process incoming data 4.4:

Listing 4.4: Materialized view for hourly aggregation

```
CREATE MATERIALIZED VIEW sensor_readings_hourly
ENGINE = SummingMergeTree()
PARTITION BY toYYYYMM(hour)
ORDER BY (device_id, sensor_type, hour)
AS SELECT
    device_id,
    sensor_type,
    toStartOfHour(timestamp) AS hour,
    avg(value) AS avg_value,
    min(value) AS min_value,
    max(value) AS max_value,
    count() AS sample_count
FROM sensor_readings
GROUP BY device_id, sensor_type, hour;
```

These materialized views allow the frontend to render historical charts with response times under one second, even when the underlying raw data table contains millions of rows. This approach required some iteration to get the performance characteristics right, but the result was well worth the effort.

4.6 Application Microservices

The Application Layer is designed following a microservices architectural pattern, ensuring modularity, scalability, and independent deployment of core functionalities. Instead of a monolithic core, the system's logic is partitioned into specialized services that communicate through well defined APIs and message queues. These microservices are responsible for orchestrating the flow of data from the LoRaWAN network to the end-user, managing the lifecycle of IoT assets, and ensuring secure access to the platform's resources. This section details the specific roles and responsibilities of each service within the solution's ecosystem.

4.6.1 Assets Manager

The Assets Manager service is responsible for tracking all devices and gateways in the system. It synchronizes with ChirpStack's API in order to import device metadata and provides a REST API through which the frontend can query and manage assets. Its functionality includes listing devices by application, tenant, or location, registering new devices in both ChirpStack and the local database, tracking device activity based on the time of the last received uplink, and managing the mapping between devices and their physical locations on campus.

4.6.2 Topics Manager

The Topics Manager is the service that orchestrates the distributed bridge architecture. It maintains a registry of active bridge instances along with the device assignments for each instance, distributes device topics across bridge instances for the purpose of load balancing, creates and configures the necessary message broker resources, whether Kafka topics or RabbitMQ bindings, and handles bridge instance failures by reassigning the affected device topics to other running instances. The reassignment logic could be improved for deployments that involve a larger number of bridge instances, but for the current scale it has proven to be sufficient.

4.6.3 Users Manager

The Users Manager handles authentication and authorization for the system. It stores user accounts and API tokens in PostgreSQL and provides endpoints for login, token generation, and role verification. Integration with the university's LDAP directory service was planned but was not included in the current implementation.

4.6.4 Frontend Application

The frontend is a Next.js application that provides a web interface for interacting with the monitoring system. It includes dashboard views that display current sensor readings through gauges, charts, and maps showing device locations across the campus. Historical data visualization is provided through time-series charts that query ClickHouse's aggregated views, ensuring that rendering remains responsive even for long time ranges. A device management section allows users to view device details, check connectivity status, and manage configurations. Alert management functionality allows users to configure threshold-based alerts that trigger notifications when sensor readings exceed or fall below specified limits.

4.7 Deployment Strategy

To ensure consistency across development, testing, and production environments, the system adopts a container-based deployment strategy using **Docker** and **Docker Compose**. This approach abstracts the underlying infrastructure, allowing each microservice and its dependencies to be packaged into isolated units that can be deployed reliably on any host system. The orchestration of these containers is managed through a multi-layer configuration that defines the network topology, data persistence volumes, and service interdependencies.

4.7.1 Containerization

Every component in the system runs inside a Docker container. Each service has a DockerFile that specifies how the image is built, what dependencies it requires, and how the service should run. A Docker Compose file was used to orchestrate the entire stack during development and testing, defining the network topology, service dependencies, and the locations of persistent data on disk [Merkel, 2014].

For the bridge instances, multi-stage Docker builds were used in order to keep the final image size small. The first stage compiles the Rust binary with all the necessary build dependencies, and the second stage copies only the compiled binary into a minimal base image. This approach results in a compact final image that is quick to transfer and deploy.

4.7.2 Development Configuration

When developing locally, everything runs on a single machine. This includes ChirpStack, along with its PostgreSQL and Redis components. Also included are a Mosquitto MQTT broker for ChirpStack's integration output, the message broker (either Kafka or RabbitMQ, depending on the version under test), ClickHouse in single-node mode, one or more MQTT bridge instances, and the microservices and frontend application.

This setup facilitates swift development cycles, enabling quick spin-up and shutdown of the complete stack through Docker Compose.

4.7.3 Production Considerations

For production deployment, the system was designed to run on a container orchestration platform such as Kubernetes [Burns et al., 2016]. A production deployment necessitates the message broker's operation within a clustered configuration to ensure high availability. Furthermore, deploying multiple bridge instances across diverse hosts is essential for fault tolerance. Data durability is achieved through the configuration of ClickHouse with replication. Monitoring and alerting systems must

be established for all infrastructure components, and automated backup procedures are crucial for both databases and configuration data.

A full production deployment using Kubernetes was beyond the scope of this dissertation, but the containerized architecture ensures that the transition from the development environment to a production environment is primarily a matter of configuration changes rather than a fundamental redesign.

Chapter 5

Results and Discussion

This chapter presents the evaluation of the implemented system, covering functional validation of the individual components and the end-to-end data flow, performance observations gathered during development and testing, a qualitative assessment of the architecture against the requirements established in earlier chapters, and a discussion of the lessons that emerged during the design and implementation process.

5.1 Functional Validation

The system was validated through a series of functional tests designed to verify that each component performs its intended role correctly and that the complete data flow operates as expected from device transmission to dashboard display.

5.1.1 LoRaWAN Connectivity

The LoRaWAN layer was tested by deploying sensor devices and verifying that their transmissions were received by the gateways and processed correctly by ChirpStack. Several key aspects were examined during this validation process.

I confirmed that devices using OTAA successfully completed the join procedure and received session keys from ChirpStack, after which they began transmitting data normally. I also verified the uplink reception path by confirming that sensor readings transmitted by the devices were received by the gateway, forwarded to ChirpStack, decoded using the appropriate device profile, and published to the MQTT broker with the correct topic structure. The JSON payloads observed on the broker were inspected and confirmed to match the expected format and content.

Additionally, I tested ChirpStack's adaptive data rate mechanism by observing how the system adjusted device spreading factors over successive transmissions based on the observed link quality, which confirmed that the network optimization was functioning as designed. I also set up devices within range of multiple gateways and

verified that ChirpStack correctly deduplicated the received packets and selected the reception with the best signal quality, which is a critical capability for reliable operation in environments where gateway coverage areas overlap.

5.1.2 MQTT Bridge Operation

The bridge instances were tested both individually and in the distributed configuration with multiple instances running simultaneously. Considering this, the validation focused on verifying that each bridge instance could perform its functions correctly and independently.

I confirmed that bridge instances correctly subscribed to the MQTT topics assigned by the Topics Manager and received messages only for their assigned devices. The JSON payloads received from ChirpStack were correctly parsed, and the extracted fields including DevEUI, timestamp, sensor values, and RF metadata were verified to match independently calculated values, which provided confidence that the transformation logic was working correctly. The transformed data was successfully published to the message broker with the correct routing keys in the RabbitMQ version and the correct topic names in the Kafka version. Network disruption simulations were also performed to verify that bridge instances correctly re-registered with the Topics Manager after disconnections and resumed processing their assigned device topics.

5.1.3 End-to-End Data Flow

The complete data path was validated by tracing a sensor reading from the moment of device transmission through to its appearance in the frontend dashboard.

The test proceeded as follows: a temperature sensor transmitted a reading, which was received by the gateway and forwarded to ChirpStack. ChirpStack decoded the payload and published it to MQTT. The MQTT bridge consumed the message, applied the necessary transformations, and published the result to RabbitMQ. The ClickHouse ingestion service consumed the RabbitMQ message and stored the reading in the database. Finally, the frontend dashboard displayed the updated temperature value.

This end-to-end verification confirmed that all components were correctly integrated and that data flowed through the system as designed. The total time from sensor transmission to dashboard display was consistently under five seconds in the test environment, which is well within the requirements for campus monitoring applications where near-real-time responsiveness rather than strict real-time guarantees is sufficient. The test was repeated at different times to check for variance, and the timing remained consistent across runs.

5.2 Performance Observations

This section presents the performance measurements gathered during the development and testing phases. In order to provide a systematic evaluation of the system's behaviour, several benchmark tests were defined and executed across the key components of the data pipeline.

5.2.1 Hardware used for the benchmarks

The benchmarks present in this chapter were measured using the following hardware:

- **Host machine:** Lenovo ThinkPad P14s Gen 2 - AMD Ryzen 7 Pro (8 cores, 16 threads, up to 4.4 GHz), 24 GB DDR4 RAM, 512 GB NVMe SSD, running Ubuntu with Docker.
- **Gateway:** SenseCAP M2 Multi-Platform LoRaWAN Indoor Gateway, set for the EU868 frequency band.
- **Sensors:** SenseCAP S2103 (air temperature and humidity) and SenseCAP S2105 (soil moisture and temperature), both operating as LoRaWAN Class A devices with OTAA activation.

All services ran simultaneously as Docker containers on the host machine, and the laptop was connected to mains power during all test runs to avoid CPU frequency throttling.

5.2.2 Bridge Processing Latency

The Rust-based MQTT bridge adds minimal latency to the data path. To measure it's latency a shell script was developed that publishes synthetic MQTT messages at controlled rates using mosquitto pub/sub service, replicating ChirpStack uplink JSON format. The MQTT bridge solution was then adjusted to log the relevant metrics. Table 5.1 presents the bridge processing latency measurements under different message load conditions.

Table 5.1: MQTT bridge processing latency under different load conditions.

Test Scenario	Avg Latency (ms)	P95 Latency (ms)	P99 Latency (ms)
10 messages/s (light load)	0.12	0.19	0.26
50 messages/s (moderate load)	0.15	0.23	0.32
200 messages/s (high load)	0.21	0.36	0.49
500 messages/s (stress test)	0.35	0.54	0.73
Sustained load (1 hour)	0.18	0.27	0.38

The choice of Rust contributed significantly to the low latency observed during preliminary testing. The absence of garbage collection means there are no pause events that could introduce unpredictable delays, and the asynchronous I/O model ensures efficient utilisation of system resources even when handling many concurrent connections.

Table 5.2 presents the resource consumption of the bridge under the same test conditions. This metric was measured with snapshots of CPU and memory usage captured using docker stats –no-stream against the bridge containers.

Table 5.2: MQTT bridge resource consumption under different load conditions.

Test Scenario	**CPU Usage (%)**	**Memory Usage (MB)**
10 messages/s (light load)	1.2	4.8
50 messages/s (moderate load)	3.1	5.3
200 messages/s (high load)	8.6	6.2
500 messages/s (stress test)	19.4	7.9
Sustained load (1 hour)	3.5	5.5

5.2.3 ClickHouse Query Performance

ClickHouse was evaluated for the typical query patterns used by the frontend dashboard. Table 5.3 presents the query response times for different types of operations and dataset sizes.

Table 5.3: ClickHouse query performance for typical dashboard operations.

Query Type	**Dataset Size**	**Response Time (ms)**
Last 24h readings (single device)	100K rows	8
Last 24h readings (single device)	500K rows	19
Last 24h readings (single device)	1M rows	36
Weekly aggregation (materialized view)	500K rows	11
Monthly aggregation (materialized view)	1M rows	23
Multi-device dashboard (10 devices)	500K rows	47

Note that the first five measurements are made with base on individual API queries executed in isolation, while the last scenario represents a concurrent workload generated by an active single-user dashboard.

These performance characteristics validate the choice of ClickHouse as the analytical database for this use case. The columnar storage engine's strength in aggregation queries, combined with the time-based partitioning strategy that I implemented, ensures that query performance degrades gracefully as the dataset grows over time.

5.2.4 Message Broker Comparison

Table 5.4 presents a comparison of the resource consumption and latency characteristics observed between the Kafka-based version 1 and the RabbitMQ-based version 2 of the system.

Table 5.4: Resource consumption and latency comparison between Kafka and RabbitMQ.

Metric	Kafka (v1)	RabbitMQ (v2)
Memory usage (idle)	814mb	176mb
Memory usage (under load)	1.38gb	287mb
CPU usage (under load)	11.8	4.6
Message delivery latency (avg)	14.2ms	1.3ms
Message delivery latency (P99)	46ms	4.9ms
Startup time	28s	6s
Disk usage (1 day of data)	312mb	84mb

It should be noted that these measurements reflect the specific scale of the campus deployment, which involves hundreds of devices and tens of messages per second. At orders of magnitude higher throughput, Kafka's architectural advantages in partitioning and distributed log processing would likely become more relevant. However, for the scale at which this system operates, the preliminary observations during development suggest that RabbitMQ is the more appropriate choice, which is consistent with the findings reported in the literature [Dobbelaere and Esmaili, 2017].

5.3 Architectural Assessment

This section evaluates how well the implemented system meets the requirements and design principles established in earlier chapters.

5.3.1 Scalability

The architecture supports horizontal scaling at multiple levels. Additional LoRaWAN gateways can be deployed to extend coverage without requiring changes to the software stack. Additional bridge instances can be launched and registered with the Topics Manager to handle increased device counts. Additional microservice instances can be deployed behind a load balancer to handle increased application traffic. ClickHouse can be configured with replication and sharding to distribute both the storage and query load across multiple nodes.

While these scaling capabilities have not been tested at large scale, the architectural patterns employed, including stateless services, message-based decoupling, and distributed processing through the bridge registration mechanism, are well-established

approaches to building scalable systems [Dragoni et al., 2017].

5.3.2 Modularity and Maintainability

The microservices architecture proved its value during the development process. Individual services could be developed and tested independently, and modifications to one service, such as changes to the bridge's transformation logic, did not require corresponding changes to other services. The message broker acts as a buffer between components, which means that a consuming service can be temporarily stopped for maintenance without affecting data producers.

The clear separation between the LoRaWAN layer managed by ChirpStack, the protocol bridging layer implemented in Rust, the message routing layer provided by RabbitMQ, the storage layer powered by ClickHouse, and the application layer comprising the various microservices ensures that each layer can evolve independently. For example, replacing ChirpStack with a different LoRaWAN network server would require changes only to the bridge's MQTT subscription and parsing logic, without affecting the rest of the system.

5.3.3 Interoperability

The use of open protocols including LoRaWAN, MQTT, and AMQP, together with open-source software including ChirpStack, RabbitMQ, and ClickHouse, ensures that the system can integrate with a wide range of existing and future components. New sensor types can be accommodated by configuring device profiles in ChirpStack without requiring any changes to the rest of the data pipeline. New consuming applications can be added by creating new queue bindings in RabbitMQ, which provides considerable flexibility for future extensions.

5.3.4 Cost Efficiency

The entire software stack is based on open-source components, which eliminates software licensing costs. The private LoRaWAN network avoids recurring subscription fees for device connectivity. The RabbitMQ-based architecture in version 2 runs on modest hardware, further reducing infrastructure costs. For an academic institution operating with limited budgets, this cost profile makes the system a viable option.

5.4 Lessons Learned

Several lessons emerged from the design and implementation process that may be valuable for other institutions considering similar deployments.

The initial takeaway is that the selection of technology should be dictated by practical needs, not by its potential maximum performance. The initial decision to use Kafka was based on its standing as a high-performance event streaming platform; however, the actual data volumes encountered in a campus deployment, even one of considerable size, are significantly less than those for which Kafka was originally designed. Opting for technologies that align with the actual scale of the problem, rather than a hypothetical future scale, results in systems that are both simpler and easier to maintain. This does not imply a disregard for scalability; rather, it necessitates a realistic understanding of what scalability entails within a specific context.

The second lesson is that the bridge component turned out to be more important than its position in the architecture might suggest. The MQTT bridge, sitting between ChirpStack and the message broker, is where protocol translation, data normalisation, and routing decisions all take place. A thoughtfully constructed and rigorously vetted bridge component benefits the entire architectural framework. Choosing Rust for its implementation, despite introducing some development hurdles, yielded a component that is both swift and dependable, consuming minimal resources.

The third lesson concerns multi-tenancy. Even in a single-institution deployment, the need to separate devices, applications, and data access by department or research group emerged early in conversations with potential users of the system. Designing for multi-tenancy from the beginning, through ChirpStack's organisational model and through the access control mechanisms in the application layer, avoids the need for painful retrofitting at a later stage.

The fourth lesson is that containerisation is effectively a requirement rather than an optional convenience. The number of components in a microservices-based IoT system, even a modest one, makes manual deployment impractical. Docker and Docker Compose transformed the development experience by making it possible to bring up the entire stack with a single command and by ensuring that the development environment closely matched the production target.

The fifth lesson is that documentation and monitoring should be treated as integral parts of the system rather than as additions to be addressed after the fact. Debugging issues in a distributed system is significantly more difficult than in a monolithic application, and centralised logging, health check endpoints, and clear documentation of the data flow and component interactions proved essential for diagnosing problems during development and would be even more critical in a production deployment.

Chapter 6

Conclusion and Future Work

This dissertation has presented the design, implementation, and evaluation of a LoRaWAN-based IoT monitoring system intended for smart campus environments. The work was motivated by the need to create a platform that is scalable, modular, and cost-effective, capable of collecting, processing, and visualising environmental data across a university campus while relying exclusively on open-source technologies and avoiding dependency on commercial platforms or recurring service subscriptions that would place an ongoing burden on institutional budgets.

The research began with a comprehensive review of the state of the art, covering fundamental IoT concepts, the smart city and smart campus paradigms, and the various communication technologies available for large-scale sensor deployments. In the context of LPWAN technologies, I examined LoRaWAN, Sigfox, and NB-IoT in a comparative analysis that provided solid justification for selecting LoRaWAN as the communication layer for this work. The factors that determined this choice include the support for private network deployment, the absence of recurring per-device costs, the energy efficiency that enables multi-year battery lifetimes for sensor nodes, and the open ecosystem that prevents vendor lock-in.

The system architecture was documented in two iterations, and this architectural progression constitutes a key contribution of this research. The initial design, designated version 1, utilized Apache Kafka as the core event streaming platform, which offered robust data persistence and high-throughput processing capabilities. Nevertheless, practical experience gained during the development phase revealed that Kafka's operational intricacies and resource demands were excessive for a campus-scale deployment. Consequently, the architecture was revised in version 2, substituting Kafka for RabbitMQ and augmenting the Rust-based middleware that connects ChirpStack's MQTT output to the message broker. The updated version preserves the core design principles specifically, separation of concerns, microservices decomposition, and event-driven communication while achieving a simpler, more lightweight, and operationally more manageable system.

The implementation covered the full technology stack, including ChirpStack config-

uration for the EU868 frequency band with multi-tenancy support, the Rust-based MQTT bridge with its distributed processing model, RabbitMQ message routing, ClickHouse analytical storage with materialized views for pre-computed aggregations, and a set of application microservices encompassing asset management, topic management, user management, and a Next.js-based frontend. All components were containerised with Docker in order to ensure consistent deployment across environments.

The functional validation demonstrated that the end-to-end data flow works correctly, from sensor transmission through LoRaWAN to processing by ChirpStack, protocol bridging, message routing, storage in ClickHouse, and display in the frontend dashboard. On the performance side, the Rust-based bridge introduces submillisecond latency, ClickHouse handles typical dashboard queries in under 50 milliseconds, and the RabbitMQ-based architecture consumes significantly fewer resources than the Kafka-based version that preceded it.

6.1 Limitations

It is important to acknowledge the limitations of this work in order to provide an accurate assessment of what was accomplished and what remains to be done.

The testing was conducted at a limited scale, with the system validated using a development environment and a modest number of devices. While the architecture is designed in order to scale to hundreds or thousands of devices, I did not have the opportunity to verify the system's behaviour under real high-load conditions. A campus-wide deployment would be necessary to evaluate the scaling characteristics under actual operational conditions.

The system was developed and tested primarily in a development environment using Docker Compose. A full production deployment with clustering, redundancy, automated failover, and comprehensive monitoring infrastructure was not carried out, as this was beyond what could reasonably be accomplished within the scope of this dissertation.

Integration with the university's identity management system through LDAP was planned but was not implemented in the current version. The Users Manager currently relies on its own user database, which would need to be integrated with institutional authentication systems before a production deployment.

The performance measurements reported in this work are based on informal observations during development rather than on rigorous benchmarking under controlled conditions. A systematic evaluation with well-defined workloads and statistical analysis would provide more reliable performance characterisation.

Finally, I have not operated the system over extended periods, which means that I do not yet have insights regarding long-term reliability, maintenance requirements,

or the effectiveness of data management strategies such as retention policies and data archiving at scale.

6.2 Future Work

Several directions for future research and development emerge naturally from the work presented in this dissertation.

The next step involves deploying the system across the Universidade de Évora campus. This will include a representative set of sensor devices in various building types and outdoor areas. This deployment will provide real-world data on network coverage, battery life, system reliability, and user satisfaction. As a result, a more complete evaluation of the architecture's performance can be conducted.

For applications that require faster response times or reduced bandwidth to the central infrastructure, edge computing nodes could be deployed alongside LoRaWAN gateways. These nodes would perform local data processing, filtering, and anomaly detection before forwarding the relevant data to the central platform, which would be particularly valuable for safety-critical applications requiring immediate local response.

The time-series data that the system collects is sufficiently rich for the application of machine learning techniques. Anomaly detection, equipment failure prediction, and the identification of patterns in energy consumption or space utilisation are all feasible applications. Integrating machine learning pipelines with the existing data infrastructure represents a natural extension of the current work.

Integrating the monitoring platform with the university's current building management and HVAC systems would facilitate closed-loop control. This would allow sensor data to automatically adjust heating, cooling, and ventilation. The potential for energy savings and enhanced occupant comfort would likely outweigh the effort needed for integration.

A mobile application would make the system more accessible to campus users, allowing them to check environmental conditions, receive alerts, and interact with the system from their smartphones. Push notifications for alert conditions would be particularly useful in a campus context.

Evaluating performance systematically, using controlled conditions and specific workloads that simulate different deployment sizes, would provide useful data for planning capacity. This approach would also allow for a more thorough comparison of the Kafka and RabbitMQ versions when they are tested under the same conditions.

While the system incorporates security measures at each layer, a comprehensive security audit and penetration testing exercise would be necessary to identify any vulnerabilities that need to be addressed before a production deployment that handles potentially sensitive data.

Ultimately, by releasing the system as an open-source project, complete with comprehensive documentation, deployment instructions, and sample configurations, other institutions could then implement and customize the platform to suit their specific campus monitoring requirements. This would, in turn, broaden the influence of this research beyond the Universidade de Évora, thus contributing to the wider smart campus research community.

6.3 Final Remarks

This work has demonstrated that it is feasible to build a smart campus monitoring system using open-source technologies and modest infrastructure. The combination of LoRaWAN for long-range, low-power wireless connectivity with a microservices-based backend architecture provides a flexible and scalable platform that can grow together with the institution's evolving needs.

The architectural evolution from Kafka to RabbitMQ illustrates an important principle in systems engineering that deserves more attention than it typically receives: the best technology for a given component is not always the most powerful or feature-rich one, but rather the one that best fits the actual requirements and constraints of the deployment context. For campus-scale IoT systems, where data volumes are moderate and operational simplicity is highly valued, a pragmatic approach to technology selection yields better outcomes than an approach driven primarily by theoretical capability.

Employing Rust for the protocol bridging middleware was a judicious choice, given its provision of the requisite performance and reliability attributes for a component situated within the critical data path. Furthermore, Rust's ownership model, which offers compile-time safety assurances, mitigates the potential for runtime errors in a production environment; this is especially advantageous in systems designed for continuous operation with minimal maintenance.

The system's design offers a robust framework for future development, encompassing the integration of supplementary sensor modalities, the introduction of novel application services, the incorporation of machine learning capabilities, and the potential for campus-wide implementation. This modular architecture facilitates the independent advancement of each of these enhancements, thereby obviating the need for a comprehensive redesign of the underlying platform.

Appendix A

MQTT Bridge Configuration Reference

This appendix provides a reference for the configuration parameters of the MQTT bridge component.

Environment Variables

Variable	Default	Description
`BRIDGE_ID`	(auto)	Unique identifier for this bridge instance
`MQTT_HOST`	localhost	ChirpStack MQTT broker hostname
`MQTT_PORT`	1883	ChirpStack MQTT broker port
`BROKER_HOST`	localhost	Message broker hostname
`BROKER_PORT`	5672	Message broker port (5672 for RabbitMQ, 9092 for Kafka)
`TOPICS_MANAGER_URL`	–	URL of the Topics Manager service
`HTTP_PORT`	8080	Port for the bridge HTTP API
`LOG_LEVEL`	info	Logging verbosity (debug, info, warn, error)

HTTP API Endpoints

Method	Path	Description
GET	`/health`	Health check endpoint
GET	`/status`	Bridge status and assigned topics
POST	`/topics/assign`	Receive topic assignment from Topics Manager
POST	`/topics/revoke`	Revoke topic assignment

Appendix B

Docker Compose Configuration

This appendix provides the Docker Compose configuration used for the development environment, illustrating the complete service stack and its interdependencies.

Listing B.1: Docker Compose services overview (simplified)

```yaml
services:
  # LoRaWAN Network Server
  chirpstack:
    image: chirpstack/chirpstack:4
    depends_on:
      - postgres
      - redis
      - mosquitto
    ports:
      - "8080:8080"

  chirpstack-gateway-bridge:
    image: chirpstack/chirpstack-gateway-bridge:4
    ports:
      - "1700:1700/udp"

  # Supporting services for ChirpStack
  postgres:
    image: postgres:15-alpine
    volumes:
      - pgdata:/var/lib/postgresql/data

  redis:
    image: redis:7-alpine

  mosquitto:
    image: eclipse-mosquitto:2
    ports:
      - "1883:1883"

  # Message Broker (v2)
```

```
  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"
      - "15672:15672"

  # Analytical Database
  clickhouse:
    image: clickhouse/clickhouse-server:latest
    ports:
      - "8123:8123"
      - "9000:9000"
    volumes:
      - chdata:/var/lib/clickhouse

  # MQTT Bridge instances
  mqtt-bridge-1:
    build: ./mqtt_bridge
    environment:
      - BRIDGE_ID=bridge-1
      - MQTT_HOST=mosquitto
      - BROKER_HOST=rabbitmq

  mqtt-bridge-2:
    build: ./mqtt_bridge
    environment:
      - BRIDGE_ID=bridge-2
      - MQTT_HOST=mosquitto
      - BROKER_HOST=rabbitmq

  # Application services
  assets-manager:
    build: ./services/assets-manager
    depends_on:
      - clickhouse

  topics-manager:
    build: ./services/topics-manager
    depends_on:
      - clickhouse
      - rabbitmq

  users-manager:
    build: ./services/users-manager
    depends_on:
      - postgres-app

  postgres-app:
    image: postgres:15-alpine
    volumes:
      - pgappdata:/var/lib/postgresql/data
```

```
83
84    frontend:
85      build: ./frontend
86      ports:
87        - "3000:3000"
88
89  volumes:
90    pgdata:
91    pgappdata:
92    chdata:
```

Note: This is a simplified version of the actual configuration. Production deployments would include additional configuration for networking, resource limits, health checks, and environment-specific parameters.

Bibliography

3GPP (2016). Cellular system support for ultra-low complexity and low throughput Internet of Things (CIoT). Technical Report TR 45.820 v13.1.0, 3GPP.

Ab Rahman, A. B. (2015). Comparison of Internet of Things (IoT) data link protocols. *Technical Report, Washington University in St. Louis*. Written under the guidance of Prof. Raj Jain. Available at: http://www.cse.wustl.edu/~jain/cse570-15/ftp/iot_dlc/.

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., and Ayyash, M. (2015). Internet of Things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials*, 17(4):2347–2376.

Alvear-Puertas, V., Rosero-Montalvo, P., Peluffo-Ordóñez, D., and Pijal-Rojas, J. (2022). Internet of Things and AI-driven air quality monitoring in smart cities: State of the art and research challenges. *International Journal of Environmental Research and Public Health*, 19(23):15888.

Augustin, A., Yi, J., Clausen, T., and Townsley, W. M. (2016). A study of LoRa: Long range & low power networks for the Internet of Things. *Sensors*, 16(9):1466.

Balasubramanian, A., Baranowski, M. S., Burtsev, A., Panda, A., Rakamari, Z., and Ryzhyk, L. (2017). System programming in Rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 156–161. ACM.

Barez, F., Bilokon, P., and Xiong, R. (2023). Benchmarking specialized databases for high-frequency data. *The Journal of FinTech*. arXiv:2301.12561.

Brocaar, O. (2023). ChirpStack open-source LoRaWAN network server. Available at: https://www.chirpstack.io/docs/.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, Omega, and Kubernetes. *ACM Queue*, 14(1):70–93.

ClickHouse Inc. (2023). ClickHouse documentation. Available at: https://clickhouse.com/docs.

Costanzo, M., Rucci, E., Naiouf, M., and De Giusti, A. (2021). Performance vs programming effort between Rust and C on multicore architectures: Case study in N-body. In *Proceedings of the 50th Argentine Conference on Informatics (JAIIO)*, pages 13–27.

Dobbelaere, P. and Esmaili, K. S. (2017). Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations. *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 227–238.

Domínguez-Bolaño, T., Barral, V., Escudero, C. J., and García-Naya, J. A. (2024). An IoT system for a smart campus: Challenges and solutions illustrated over several real-world use cases. *Sensors*, 24(10):3150.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. *Present and Ulterior Software Engineering*, pages 195–216.

Farahani, S. (2011). *ZigBee Wireless Networks and Transceivers*. Newnes.

Fortes, S., Santoyo-Ramón, J. A., Palacios, D., Baena, E., Mora-García, R., Medina, M., Mora, P., and Barco, R. (2019). The campus as a smart city: University of Málaga environmental, learning, and research approaches. *Sensors*, 19(6):1349.

Gomez, C., Oller, J., and Paradells, J. (2012). Overview and evaluation of Bluetooth Low Energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753.

Haxhibeqiri, J., De Poorter, E., Moerman, I., and Hoebeke, J. (2018). A survey of LoRaWAN for IoT: From technology to application. *Sensors*, 18(11):3995.

Hiertz, G. R., Denteneer, D., Max, S., Taori, R., Cardona, J., Berlemann, L., and Walke, B. (2010). IEEE 802.11s: The WLAN mesh standard. *IEEE Wireless Communications*, 17(1):104–111.

Hohpe, G. and Woolf, B. (2004). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

Jung, R., Jourdan, J.-H., Krebbers, R., and Dreyer, D. (2021). Safe systems programming in Rust. *Communications of the ACM*, 64(4):144–152.

Khan, R., Khan, S. U., Zaheer, R., and Khan, S. (2012). Future Internet: The Internet of Things architecture, possible applications and key challenges. *Proceedings of the 10th International Conference on Frontiers of Information Technology*, pages 257–260.

Klabnik, S. and Nichols, C. (2023). *The Rust Programming Language*. No Starch Press, 2nd edition.

Kreps, J., Narkhede, N., and Rao, J. (2011). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB Workshop*, pages 1–7.

LoRa Alliance (2020). LoRaWAN 1.0.4 specification. Technical report, LoRa Alliance Technical Committee.

Mekki, K., Bajic, E., Chaxel, F., and Meyer, F. (2019). A comparative study of LPWAN technologies for large-scale IoT deployment. *ICT Express*, 5(1):1–7.

Merkel, D. (2014). Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

Mikhaylov, K., Moiz, R., Pouttu, A., Rapún, J. M. M., and Gascon, S. C. (2021). LoRaWAN multi-year operation: Lessons learned from the large-scale indoor deployment. *IEEE Internet of Things Journal*, 8(7):5760–5774.

Muhamad, W. N. A. W., Kurniawan, N. F. S., Kumar, D., Suhaimy, S. J., and Zaman, F. K. B. (2017). Smart campus concept for the students of higher learning institution. *International Journal of Applied Engineering Research*, 12(24):15183–15190.

Naik, N. (2017). Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. *Proceedings of the IEEE International Systems Engineering Symposium*, pages 1–7.

Naqvi, S. N. R., Yfantidou, S., and Zimányi, E. (2017). Time series databases and InfluxDB. In *Studienarbeit, Université Libre de Bruxelles*.

Narkhede, N., Shapira, G., and Palino, T. (2017). *Kafka: The Definitive Guide*. O'Reilly Media.

Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media.

Noura, M., Atiquzzaman, M., and Gaedke, M. (2019). Interoperability in Internet of Things: Taxonomies and open challenges. *Mobile Networks and Applications*, 24(3):796–809.

OASIS Standard (2019). MQTT version 5.0. Available at: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.

Ortiz, G., Castillo, M., García-de Prado, A., and Boubeta-Puig, J. (2022). A microservice architecture for real-time IoT data processing: A reusable Web of Things approach for smart ports. *Computer Standards & Interfaces*, 81:103604.

Raza, U., Kulkarni, P., and Sooriyabandara, M. (2017). Low power wide area networks: An overview. *IEEE Communications Surveys & Tutorials*, 19(2):855–873.

Roman, R., Zhou, J., and Lopez, J. (2013). On the features and challenges of security and privacy in distributed Internet of Things. *Computer Networks*, 57(10):2266–2279.

Sanchez-Iborra, R. and Cano, M.-D. (2016). State of the art in LP-WAN solutions for industrial IoT services. *Sensors*, 16(5):708.

Semtech Corporation (2021). LoRa and LoRaWAN: A technical overview. Technical report, Semtech. White Paper.

Spachos, P. (2020). Towards LoRa enabled IoT campus buildings with air quality monitoring. *Proceedings of the IEEE International Conference on Communications Workshops*, pages 1–6.

Tokio Contributors (2023). Axum: Ergonomic and modular web framework built with Tokio, Tower, and Hyper. Available at: https://github.com/tokio-rs/axum.

Vercel (2023). Next.js documentation. Available at: https://nextjs.org/docs.

Videla, A. and Williams, J. J. W. (2012). *RabbitMQ in Action: Distributed Messaging for Everyone*. Manning Publications.

Villegas-Ch, W., Palacios-Pacheco, X., and Luján-Mora, S. (2019). Application of a smart city model to a traditional university campus with a big data architecture: A sustainable smart campus. *Sustainability*, 11(10):2857.

Wang, Y. C. and Chen, T.-H. (2017). A survey of NB-IoT in 5G. *Proceedings of the IEEE International Conference on Applied System Innovation*, pages 1266–1268.

Zanella, A., Bui, N., Castellani, A., Vangelista, L., and Zorzi, M. (2014). Internet of Things for smart cities. *IEEE Internet of Things Journal*, 1(1):22–32.