Development of server side application for smart waste systems

Soheyb Merah - a53181

Report presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:
Rui Pedro Lopes

Bragança
October, 2025

Development of server side application for smart waste systems

Soheyb Merah - a53181

Report presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:
Rui Pedro Lopes

Bragança
October, 2025

Dedication

To my family, whose unwavering support and love have been my foundation.

To my dear friends, for their encouragement and companionship throughout this journey.

To **Mr. Kefah**, for his generous support and understanding, allowing me the time and space to focus on this work.

To **Rui Pedro**, my supervisor, for his mentorship, guidance, and constant encouragement throughout the research process.

To Professor **Ahmed Abbache** from Chlef University, whose early academic influence played a significant role in shaping my academic path.

And most of all, to my beloved wife **Amira**, your patience, strength, and love have been my greatest motivation. This achievement is shared with you.

Acknowledgment

I would like to express my sincere gratitude to my supervisor, Professor Rui Pedro, for his valuable guidance, continuous support, and insightful feedback throughout the course of this research. His expertise and encouragement were essential to the successful completion of this thesis.

This work was developed within the project "RAICYCLE - Advanced Technologies for Smart Waste Processing and Recycling", with the reference 2024.07316.IACDC and DOI 10.54499/2024.07316.IACDC.

Gostaria de expressar a minha sincera gratidão ao meu orientador, Professor Rui Pedro, pela orientação valiosa, apoio contínuo e contributos esclarecedores ao longo deste trabalho de investigação. A sua experiência e incentivo foram essenciais para a realização desta dissertação com sucesso.

Este trabalho foi desenvolvido no âmbito do projeto "RAICYCLE - Tecnologias Avançadas para o Processamento e Reciclagem Inteligentes de Resíduos", com a referência 2024.07316.IACDC e DOI 10.54499/2024.07316.IACDC.

Abstract

This thesis presents the design and implementation of a scalable backend server for real-time monitoring and analytics of smart waste management systems. The server collects data from decentralized sensors installed in urban dumpsters for detection and classification of volatile organic compounds through an electronic nose sensor. Upon detecting hazardous gas thresholds, the system generates automated alerts to waste-collection teams, thereby improving worker safety and reducing environmental risks.
Additionally, the backend provides a web-based dashboard for visualizing historical and live analytics, enabling municipal authorities to optimize collection routes, predict fill-level trends, and reduce operational costs.
The architecture leverages a microservices approach with RESTful APIs, a time-series database for sensor data storage, and message-queueing for event-driven notifications.
The proposed solution contributes to the fields of Internet of Things (IoT) in smart cities, environmental monitoring, and backend systems engineering.

Keywords: Smart waste management; Internet of Things; back-end server; real-time monitoring; gas detection; data analysis; hazardous gas alert.

Resumo

Esta tese apresenta o projeto e implementação de um servidor backend escalável para monitorização e análise em tempo real de sistemas de gestão de resíduos inteligentes. O servidor recolhe dados de sensores descentralizados instalados em contentores de lixo urbanos, medindo parâmetros como a concentração de gases, níveis de enchimento, temperatura e humidade. Ao detetar limites de gases perigosos, o sistema gera alertas automatizados para as equipas de recolha de resíduos, melhorando a segurança dos trabalhadores e reduzindo os riscos ambientais.

Além disso, o backend disponibiliza um painel web para visualização de análises históricas e em tempo real, permitindo às autoridades municipais otimizar as rotas de recolha, prever as tendências dos níveis de enchimento e reduzir os custos operacionais.

A arquitetura utiliza uma abordagem de microsserviços com APIs RESTful, uma base de dados de séries temporais para armazenamento de dados de sensores e enfileiramento de mensagens para notificações orientadas por eventos.

A solução proposta contribui para as áreas de Internet das Coisas (IoT) em cidades inteligentes, monitorização ambiental e engenharia de sistemas backend.

Keywords: Gestão inteligente de resíduos; Internet das Coisas, servidor back-end, monitorização em tempo real, deteção de gás, análise de dados, alerta de gases perigosos.

Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.2 Objectives . . . 2

1.3 Structure . . . 3

2 Context and Technologies **5**

2.1 Technology Stack . . . 6

2.1.1 Python . . . 6

2.1.2 FastAPI . . . 7

2.1.3 MongoDB . . . 7

2.1.4 MQTT . . . 8

2.1.5 Keycloak . . . 9

2.2 State of the art . . . 10

2.3 Summary . . . 16

3 Methodology **17**

3.1 Approach . . . 17

3.2 Methodology . . . 19

3.3 Functional Requirements . . . 20

3.4 Non-Functional Requirements . . . 22

3.5 Summary . . . 23

4 Development **25**

4.1 Development Overview . . . 25

4.1.1 Objectives . . . 25

4.1.2 Methodology . . . 27

4.1.3 Key Design Choices . . . 28

4.2 System Architecture . . . 30

4.2.1 Overview . . . 30

4.2.2 Web Layer . . . 32

4.2.3 Identity and Access Management . . . 33

4.2.4 Data Ingestion and Storage . . . 35

4.2.5 Messaging and Events . . . 36

4.2.6 Configuration and Environments . . . 37

4.3 Endpoints . . . 38

4.4 Packages and Dependencies . . . 40

4.5 Notable Implementation Details . . . 41

4.6 Summary . . . 42

5 Tests and Discussion **43**

5.1 Swagger . . . 43

5.2 Benchmarking . . . 45

5.2.1 Introduction . . . 45

5.2.2 Benchmark test . . . 46

5.2.3 Results . . . 49

6 Conclusions **55**

Bibliography **57**

List of Tables

4.1 API Endpoints Reference 40

5.1 Hypercorn workers benchmarking 50
5.2 Simplified benchmark overview 50

List of Figures

2.1 Python . . . 6
2.2 FastAPI . . . 7
2.3 MongoDB . . . 8
2.4 MQTT . . . 9
2.5 Keycloak . . . 10

3.1 Project overview . . . 19
3.2 System Architecture Overview . . . 20
3.3 Detailed project diagram . . . 23

5.1 High-level overview of the API endpoints . . . 44
5.2 Data schemas for various API requestss . . . 44
5.3 Detailed view of the user login endpoint . . . 45
5.4 Total request time per #workers . . . 51
5.5 Minimum request time per #workers . . . 52
5.6 Maximum request time per #workers . . . 52
5.7 Average request time per #workers . . . 53

Acronyms

API Application Programming Interface.

ARIMA Autoregressive Integrated Moving Average.

ASGI Asynchronous Server Gateway Interface.

DB Database.

GRU Gated Recurrent Unit.

HTTP Hypertext Transfer Protocol.

IAM Identity and Access Management.

IoT Internet of Things.

LoRaWAN Long Range Wide Area Network.

MAE Mean Absolute Error.

MQTT Message Queuing Telemetry Transport.

OAuth OAuth 2.0.

OIDC OpenID Connect.

PSO Particle Swarm Optimization.

QoS Quality of Service.

R^2 Coefficient of Determination.

REST Representational State Transfer.

RMSE Root Mean Square Error.

SAML Security Assertion Markup Language 2.0.

SPARS SPARS Prediction Model.

SPRUCE Spruce and Peatland Responses Under Changing Environments.

SSF Spark Streaming framework.

SSO Single Sign-On.

SVM Support Vector Machine.

SWOT Strengths, Weaknesses, Opportunities, and Threats.

TRIAS TRIAS Initiative.

VOC Volatile Organic Compound.

Chapter 1

Introduction

Cities around the world are increasingly adopting smart city technologies to optimize municipal services in response to rapid urbanization and escalating environmental concerns. In addition to being a major operational expense, waste management poses significant risks to both the environment and public health when inefficiencies, hazardous conditions, or improper disposal practices go undetected.

Conventional collection methods rely on fixed routes and schedules, which often result in either under serviced or over serviced containers. Without timely intervention, overfilled dumpsters may accumulate harmful byproducts such as hydrogen sulfide and methane, endangering nearby residents and sanitation workers. Moreover, the absence of real time monitoring facilitates the illegal dumping of dangerous or prohibited waste, further exacerbating health and safety risks within urban areas.

By integrating Internet of Things (IoT) sensors into waste management infrastructure, cities can transition from reactive to proactive operations. Such systems enable the early detection of hazardous conditions, unauthorized waste disposal, and volatile compounds, thereby enhancing sustainability, public safety, and operational efficiency.

1.1 Motivation

Although sensor hardware and low-power wireless networks have matured, there remains a gap in robust, end-to-end backend solutions that can ingest high-frequency data from widely distributed devices, process it in real time, and present actionable insights. Key motivating factors for this research include:

- **Worker Safety:** Automated detection and classification of Volatile Organic Compound (VOC) can assist in identifying types of waste, thereby preventing exposure incidents and enabling timely evacuations.
- **Operational Efficiency:** Real-time, fill-level and content type monitoring enables dynamic route planning, reducing fuel consumption and labor costs.
- **Data-Driven Decision-Making:** Historical analytic support trend forecasting and capacity planning for municipal authorities.
- **Scalability and Resilience:** Designing a system capable of handling thousands of concurrent sensor updates with minimal latency is critical for city-wide deployments.

1.2 Objectives

This work aims to design, implement, and evaluate a backend server that:

- Reliably collects and stores streaming data from decentralized dumpsters capturing gas types, fill levels, temperature, and humidity.
- Processes data in near real time to trigger automated alerts whenever predefined hazardous-gas thresholds are exceeded.
- Provides an interactive dashboard for visualizing both live sensor readings and historical trends, enabling route optimization and predictive analytic.

- Demonstrates scalability through stress testing, measuring throughput, latency, and system resilience under simulated high-volume scenarios.

1.3 Structure

This dissertation is organized into five main chapters, as follows:

- Chapter 1 introduces the work, outlining the context, research objectives, and motivation behind this study;
- Chapter 2 provides an overview of the relevant technologies and the state of the art, including the technology stack and current trends in the field;
- Chapter 3 details the research methodology, describing the adopted approach, functional and non-functional requirements, and the overall methodological framework;
- Chapter 4 presents the development process, including objectives, key design choices, system architecture, endpoints, dependencies, and notable implementation details;
- Chapter 5 focuses on system evaluation, including benchmarking tests, analysis of experimental results, and discussion of system performance.
- Chapter 6 discusses initial goals, key contributions and results.

Chapter 2

Context and Technologies

The development of modern software systems requires a careful alignment between theoretical foundations and practical implementation choices. On the one hand, it is essential to situate the problem within its scientific and industrial context, identifying existing approaches and assessing their strengths and limitations. On the other hand, the success of an implementation depends on the selection of appropriate technologies that ensure scalability, reliability, and maintainability.

This chapter is divided into two parts. The first part, *Technology Stack*, presents the main tools, frameworks, and protocols used in the design and implementation of our system. Each technology is discussed in terms of its key features and the rationale behind its selection. The second part, *State of the Art*, provides an overview of existing research and industrial solutions relevant to our domain, highlighting the advances achieved so far and the gaps that this work aims to address.

By combining a critical review of the state of the art with a clear presentation of the adopted technologies, this chapter establishes both the theoretical and practical foundations on which the proposed solution is built.

2.1 Technology Stack

This section provides a detailed presentation and justification of the primary technologies adopted for the implementation of the system. Each technology was selected according to the requirements of performance, scalability, and security.

2.1.1 Python

Python is a high level, interpreted programming language widely adopted in both academia and industry. It supports multiple paradigms, including object oriented, functional, and procedural programming, and has an extensive ecosystem of libraries and frameworks. Python's simplicity and readability make it accessible, while its versatility allows it to be applied in diverse fields ranging from web development and data science to artificial intelligence and systems programming.

The choice of Python for this project is motivated by several factors. First, it enables rapid prototyping and development thanks to its clean syntax and large standard library. Second, its ecosystem includes mature tools for networking, concurrency, and API development, which directly align with the needs of this work. Additionally, Python's large and active community ensures long term support, extensive documentation, and a wide selection of third party packages, reducing development time and effort.

Figure 2.1: Python

2.1.2 FastAPI

FastAPI is a modern, asynchronous web framework for Python designed to simplify the development of APIs. It is built on top of Starlette and Pydantic, leveraging Python type hints to provide automatic validation, serialization, and interactive API documentation. One of its most notable features is its performance, which rivals Node.js and Go frameworks, thanks to its asynchronous request handling.

FastAPI was chosen because it strikes a balance between usability and scalability. Its automatic generation of OpenAPI and Swagger documentation accelerates development and facilitates collaboration with other teams. Furthermore, its asynchronous capabilities allow the system to handle large numbers of concurrent requests efficiently, which is critical in distributed architectures. Compared to older frameworks like Flask or Django, FastAPI provides a more modern and performant solution, while remaining lightweight and easy to integrate into a microservices environment.

Figure 2.2: FastAPI

2.1.3 MongoDB

MongoDB is a NoSQL, document oriented database that stores information in a flexible, JSON like format. It supports dynamic schemas, which makes it adaptable to applications where data structures evolve over time. MongoDB also offers built in replication, sharding, and high availability, which makes it particularly well suited for distributed systems. It integrates easily with modern applications through native drivers and provides

advanced querying and aggregation capabilities.

The selection of MongoDB was based on its ability to manage heterogeneous and semi structured data without requiring predefined schemas. This flexibility reduces the overhead of schema migrations and supports rapid iterations during development. Furthermore, MongoDB's scalability and fault tolerance ensure that the system can handle growing data volumes while maintaining reliability. The ecosystem around MongoDB, including Atlas for managed services, also facilitates deployment and monitoring, reducing the operational burden on the development team.

Figure 2.3: MongoDB

2.1.4 MQTT

Message Queuing Telemetry Transport (MQTT) is a lightweight publish/subscribe messaging protocol designed for efficient communication over unreliable or constrained networks. It is widely used in IoT systems and distributed applications where devices need to exchange messages with minimal overhead. MQTT provides features such as different Quality of Service (QoS) levels, retained messages, and persistent sessions, which enhance its reliability and adaptability.

The decision to use MQTT stems from its suitability for real time communication between distributed components. Unlike traditional Hypertext Transfer Protocol (HTTP) based communication, MQTT introduces minimal latency and reduces bandwidth consumption, which is essential in systems requiring responsiveness. Its decoupled publish/subscribe model improves scalability and allows services to interact without direct

knowledge of each other. This aligns perfectly with a microservices architecture, ensuring flexibility, robustness, and extensibility.

Figure 2.4: MQTT

2.1.5 Keycloak

Keycloak is an open source Identity and Access Management (IAM) solution that provides centralized authentication and authorization. It supports a wide range of protocols, including OAuth 2.0 (OAuth), OpenID Connect (OIDC), and Security Assertion Markup Language 2.0 (SAML), making it compatible with diverse applications and services. Keycloak also provides features such as Single Sign-On (SSO), social login integration, multi factor authentication, and fine grained access control.

Keycloak was chosen to ensure strong security and centralized user management across all system components. Without Keycloak, authentication and authorization logic would need to be implemented separately in each service, increasing complexity and the risk of security vulnerabilities. By delegating identity management to Keycloak, the system benefits from a proven and actively maintained solution, reducing the burden of implementing security in house. Additionally, Keycloak's extensibility and support for federation make it a future proof choice, as it can easily integrate with external identity providers and scale with the system's requirements.

Figure 2.5: Keycloak

2.2 State of the art

Content types, fill levels, temperature, humidity, and other streaming data from decentralized dumpster sensors can be reliably collected, stored, and processed very instantly thanks to state of the art backend technologies.
In order to effectively collect sensor data and initiate automated alarms when hazardous gas thresholds are crossed, modern designs make use of stream processing frameworks, reliable storage systems, and alerting mechanisms.

The TRIAS Initiative (TRIAS) initiative aims to improve output quality affected by humidity and temperature fluctuations while adhering to cost constraints. It requires a real time monitoring and alarm system using temperature and humidity sensors. The system will trigger alarms when readings exceed set limits. Research methods included interviews, literature reviews, and observations, employing the Waterfall methodology for development and Strengths, Weaknesses, Opportunities, and Threats (SWOT) analysis for evaluation. Key findings highlight the importance of a visualization tool and cost reduction. Grafana serves as the frontend for visualization and monitoring, while Node Red functions as the backend, collecting and synchronizing sensor data in a MySQL Database (DB). Node Red also manages alert systems that send notifications when sensor readings surpass defined thresholds, enabling near real time processing [1].

The increasing production of semi structured data from sensors across various industries requires efficient processing and storage solutions. While NoSQL databases like

MongoDB and frameworks such as Hadoop have been explored, there has been limited research on integrating MongoDB with Apache Spark for handling high velocity sensor data. This study aims to compare MongoDB's performance with and without sharding when used with Spark to determine optimal configurations for managing sensor data. MongoDB's scalability and availability make it suitable for securely collecting data from decentralized sources, while Apache Spark enables near real time analysis and retrieval, crucial for automated alerts when hazardous gas levels are exceeded [2].

The traditional coal mine gas concentration prediction process faces challenges like slow data timeliness and inefficient prediction models, leading to low accuracy. To address these issues, a new method using the Spark Streaming framework is proposed, which integrates the Autoregressive Integrated Moving Average (ARIMA) model and Support Vector Machine (SVM) model into a prediction model called SPARS Prediction Model (SPARS). This model processes large batches of real time data quickly and allows for intermittent updates to enhance learning from data characteristics. By combining ARIMA for linear data and SVM for nonlinear data, the SPARS model improves prediction efficiency and timeliness. The model has been validated with on site gas data, demonstrating its capability for accurate real time predictions and providing a novel approach for gas concentration monitoring in coal mines. Spark Streaming facilitates reliable processing of large scale data, supporting timely and scalable backend operations, while the SPARS model ensures quick adaptation to new data for prompt alerts on hazardous gas levels [3].

The rise of affordable small sensors has created new possibilities for real time monitoring systems across various fields. The IoT connects diverse devices through different protocols for large scale interoperability. This paper discusses the integration of open source tools for collecting, monitoring, and processing real time data from sensors. They utilize the ThingsBoard platform for data collection and Spark Streaming for data analytics. This architecture can be applied in various monitoring scenarios, including clinical,

environmental, and energy processes, to track key parameters and issue alerts in critical situations. Specifically, the system focuses on monitoring respiratory parameters in patients with chronic respiratory diseases., enabling automatic ventilotherapy and timely alerts for doctors. ThingsBoard effectively collects and stores streaming data from decentralized sources, while Spark Streaming allows for near real time analysis and automated alerts when hazardous conditions arise. Integrating IoT devices with scalable backend frameworks enhances continuous monitoring and rapid response capabilities [4].

Recent advancements in large scale networked sensor technologies and big data computing have facilitated new applications in smart city ecosystems. This paper defines big sensor data systems and reviews their development and applications, categorizing existing research by characteristics and challenges within smart city layers. It discusses various applications of these systems and the potential of networked sensors in the big data era. The paper concludes with future work directions and potential applications, aiming to provide insights for researchers and encourage practical solutions for smart city deployment. Additionally, it highlights the capability of big sensor data systems to process diverse streaming data in near real time, enabling automated alerts for hazardous conditions. Backend architectures must tackle challenges related to data volume, velocity, and integration to ensure effective hazard detection and response [5].

Climate change studies are crucial in modern science, with experiments like the Spruce and Peatland Responses Under Changing Environments (SPRUCE) in northern Minnesota focusing on ecosystem responses to climate warming and elevated CO2 levels. This experiment generates extensive observational data, necessitating a reliable data collection system and real time monitoring capabilities. The publication discusses the establishment of a near real time data system using the PakBus protocol, which facilitates robust data transfer over satellite links. Such systems enable automated alerts by processing sensor data against hazardous gas thresholds. Effective backend architectures for climate experiments require dependable data acquisition, secure transmission, and scalable analytics

for safety interventions [6].

The prediction and early warning of mine gas concentrations are crucial for safe coal mine operations. This study introduces an early warning model using the Spark Streaming framework (SSF), which integrates a Particle Swarm Optimization (PSO) algorithm and a Gated Recurrent Unit (GRU) model. Experimental analysis optimizes model parameters, and the model's efficiency is validated through a control variable approach, with The prediction accuracy was assessed using Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and Coefficient of Determination (R^2) metrics. Results indicate high efficiency and accuracy in gas concentration prediction and warning, along with fast data processing and fault tolerance. This approach enhances the reliability of real time monitoring of various sensor data, including gas concentrations, and supports timely automated alerts for hazardous conditions, contributing to improved safety in coal mines [7].

Large volumes of data are produced by ubiquitous sensing environments, including sensor networks, and this volume will continue to expand with the advent of the IoT. Cloud computing provides scalable storage and processing capabilities for data, enabling effective aggregation and analysis from diverse sources. Nevertheless, apprehensions over privacy and data governance emerge, as data proprietors forfeit control once their information is transferred to the Cloud. A reliable Cloud architecture is essential for the secure management of sensitive information. This study examines the essential characteristics for this design and introduces the SensorCloud security architecture, which implements end to end data access control and stringent service level isolation. An assessment of an initial prototype suggests that this architecture represents a significant improvement to existing Cloud solutions, allowing data owners to retain control over their sensor data during its entire lifecycle [8].

In *The Art of Scalability*, Abbott and Fisher present one of the most comprehensive frameworks for understanding scalability from both technical and organizational perspectives. They define scalability as a multidimensional concept involving software architecture, infrastructure provisioning, and team structure. The authors introduce the "Scale Cube" model, which categorizes scaling into three dimensions—horizontal duplication of instances, functional decomposition, and data partitioning. This work serves as a foundational reference for scalable backend system design, providing a balance between theory and practical implementation guidelines 9.

Chris Richardson's *Microservices Patterns* advances the discussion by offering a pattern based approach to building scalable microservice architectures. The book explores core design challenges such as inter service communication, data consistency, and deployment automation. Richardson introduces key patterns like the Saga, Command Query Responsibility Segregation (CQRS), and Event Sourcing, which enable systems to handle scalability and fault tolerance more effectively. His work is highly regarded for bridging the gap between monolithic and microservice based architectures, particularly in cloud environments 10.

Pacheco's *Microservice Patterns and Best Practices* complements Richardson's work by emphasizing practical design patterns and implementation strategies for achieving scalable and maintainable systems. The author elaborates on techniques such as Domain Driven Design (DDD), event driven communication, and service orchestration. Moreover, the book provides best practices for testing and monitoring microservices, which are essential to maintain performance under increasing load. Pacheco's contribution is valuable for practitioners seeking hands on methodologies for backend scalability 11.

Theo Schlossnagle's *Scalable Internet Architectures* is a pioneering text that addresses scalability challenges in early large scale internet systems. Although published before the widespread adoption of cloud native paradigms, it remains influential for its focus

on performance optimization, caching strategies, and redundancy mechanisms. The book introduces techniques for managing high traffic environments and provides insights into the trade offs between scalability, availability, and maintainability. Schlossnagle's work laid the groundwork for many principles later adopted in modern distributed systems 12.

In their systematic literature review, Gurung, Shrestha, and Chulyadyo analyze scalability trends and patterns in microservice based systems. Their study categorizes scalability strategies into infrastructure level (e.g, container orchestration with Kubernetes) and architecture level approaches (e.g, event driven and serverless patterns). The review highlights the increasing reliance on observability tools and distributed tracing frameworks as core enablers of scalable systems. The authors' findings underscore a growing consensus on the hybrid use of microservices and serverless functions for elasticity and cost efficiency 13.

Shah, Jagtap, and Jain focus on scalable backend architectures in the context of mobile ecosystems. Their article, *Architecting Analytics Driven Mobile Ecosystems*, discusses the use of data pipelines, event streams, and real time analytics frameworks to support high user concurrency and low latency. The paper emphasizes the role of backend scalability in enabling adaptive user experiences and intelligent data flow. Their proposed model integrates microservices with machine learning pipelines, making it particularly relevant for data driven applications such as ride hailing or IoT based systems 14.

Krishnan's literature review on backend development optimization offers a meta analysis of techniques aimed at improving operational efficiency in large scale systems. The study identifies the convergence between DevOps practices, containerization, and infrastructure as code as key enablers of scalability. It also discusses the role of load balancing, caching, and asynchronous messaging in optimizing backend throughput. Krishnan's synthesis is useful for understanding the operational dimensions of scalability and the alignment between software engineering and infrastructure management 15.

2.3 Summary

This chapter presents the theoretical and technological foundations underpinning the design and implementation of our system. The analysis commenced with a comprehensive presentation of the technology stack, wherein each tool, framework, and protocol was elucidated and validated concerning its pertinence to the project's specifications. This was succeeded by an examination of the current state of the art, offering a summary of existing research and industrial solutions in the domain, while pinpointing the opportunities and difficulties that inspired our methodology.

By integrating knowledge of contemporary advancements with a thoughtfully selected array of technology, this work establishes a robust foundation for the subsequent phases of the project. The knowledge gathered from the technology assessment and the literature research not only informs the architectural choices but also shows how the suggested solution fits into the larger scientific and technological context.

Chapter 3

Methodology

This chapter delivers a comprehensive analysis of the problem and outlines the proposed solution.

In the case of software development projects, there should include tools and concepts related to the modeling and analysis (such as UML diagrams or others). There should also describe the tasks that the system should implement and the authors that interact with it. The description should be detailed to understand the difficulties associated to the problem resolution.

3.1 Approach

Content type and filling level are the two main environmental characteristics monitored by embedded sensors installed in each trash collection container. The system is designed to detect dangerous or prohibited waste, such as used oils, burning ash, or other hazardous materials. Gas sensors continuously monitor the presence of harmful pollutants, including methane, carbon dioxide, and hydrogen sulfide, which result from waste decomposition. Monitoring these gases enhances environmental safety through early detection of aberrant emissions and yields significant data for comprehending the chemical dynamics within the waste. Simultaneously Esp32 Huzzah, Bme688 development kit, ToF and grove VL53L0X, for GPS, Grove air530 and Lora grove-wio-e5 are employed to ascertain the fill level of

each container, providing real time data on garbage accumulation rates and enhancing collection logistics.

By using the MQTT protocol to transfer the gathered data, the sensor nodes are made to function in an ecosystem that is enabled by the IoT. MQTT is ideally suited for this application due to its lightweight architecture and minimal bandwidth requirements, rendering it effective for devices reliant on limited power sources such as batteries or solar panels. Each sensor node periodically transmits data packets to a proximate MQTT gateway, which serves as an intermediate between the dispersed sensor network and the central server architecture. This design decision guarantees scalability, allowing for the seamless integration of more dumpsters into the system with little reconfiguration.

The server infrastructure is essential for processing and managing sensor data. The server functions as a MQTT client that subscribes to pertinent topics provided by the gateway. As sensor nodes transmit messages, the server monitors them in real time and promptly analyzes the payloads, collecting essential data such as gas concentrations, waste fill levels, and timestamps. The parsing process may include data validation procedures to eliminate erroneous or inconsistent readings, thus enhancing the reliability of the stored information.

Upon successful processing, the data is methodically stored in a relational or time series database. The selection of database technology is contingent upon the anticipated query types; for example, a time series database like InfluxDB may prove very effective for managing frequent sensor changes, facilitating swift retrieval and analysis of temporal trends. In contrast, a relational database offers a solid schema design and facilitates integration with other enterprise systems. The system establishes a basis for advanced analytics by storing data in a structured and queryable format, enabling predictive modeling of trash accumulation, anomaly detection in gas emission patterns, and optimization of collection vehicle routes.

The following figure 3.1 displays the project overview in a brief way:

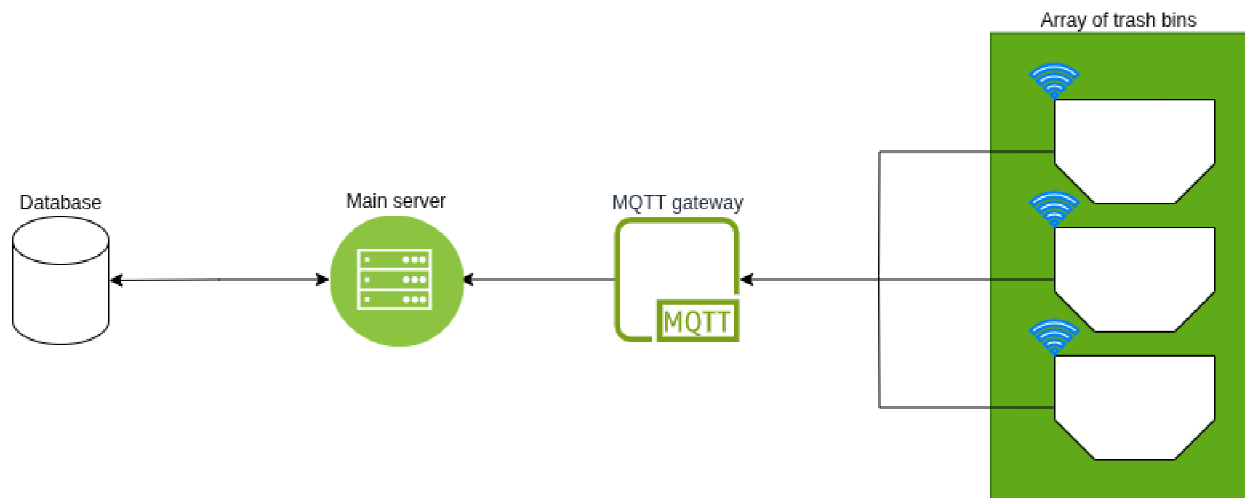

Figure 3.1: Project overview

3.2 Methodology

The system incorporates protocols, data processing pipelines, and user facing components to guarantee the efficient, reliable, and secure management of sensor data, hence facilitating operational functionality and strategic decision making.

Note: A key focus of the methodology is to address several core challenges:

- **Data communication** from multiple sensors, managed using the MQTT protocol for lightweight and scalable message delivery.
- **Data representation** through JSON for human readable formatting and interoperability between services.
- **Data serialization** with Protobuf (Protocol Buffers) for efficient and compact data transfer.
- **Data storage** in NoSQL databases to handle high volume and variable structure of sensor data.

- **Data processing and access** via a dedicated service tier and Representational State Transfer (REST) API, ensuring modular, scalable, and secure access to processed information.

The following cycle in figure 3.2 illustrates the overall system architecture. Data from embedded sensors serve as the primary sources, transmitting environmental measurements and status information to the system. Communication is handled via an MQTT broker, ensuring reliable, low latency message delivery.
Data are serialized using Protobuf for efficient transmission and stored in a NoSQL database, enabling flexible handling of heterogeneous and semi structured datasets. The service tier processes the data, performing necessary transformations, analyses, and aggregation. Finally, a REST API provides secure and structured access for users and client applications, completing the end-to-end data flow from acquisition to consumption.

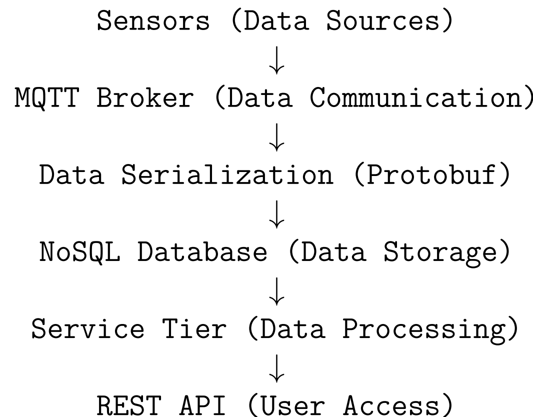

Figure 3.2: System Architecture Overview

3.3 Functional Requirements

The system must employ Protocol Buffers (Protobuf) as the serialization standard for data transmission. In contrast to JSON or XML, Protobuf produces compact, binary encoded messages, significantly reducing bandwidth usage and decreasing transmission latency.

This is particularly critical in IoT environments, where devices are resource constrained and communication channels may be unreliable. The strongly typed schema offered by Protobuf minimizes data discrepancies between the publisher (sensor node) and the subscriber (server), facilitating rapid data parsing and maintaining backward compatibility as message definitions evolve.

The system must include an MQTT listener on the server that continuously decodes incoming Protobuf messages, validates their integrity, and stores them in the database. This ensures real time data ingestion while sustaining high throughput — an essential capability given the potential scale of distributed dumpsters in urban deployments. Furthermore, the system must support both reactive decision making (e.g., triggering alerts upon threshold breaches) and retrospective analysis (e.g., examining waste generation trends over time). Querying data within defined date ranges should also be supported to facilitate operational reporting, environmental compliance assessments, and predictive modeling.

Beyond raw data storage, the system must support an annotation mechanism that allows administrators or analysts to categorize sensor readings within specified time ranges. This feature adds semantic context such as anomalous activity, external interventions, or scheduled data collection to raw datasets. Such enriched data enable more advanced analyses, including machine learning model training, anomaly detection, and operational audits, thereby transforming the database from a simple data repository into a structured knowledge source.

A web based dashboard must be provided to enable dynamic configuration of operational thresholds. For example, filling level thresholds can trigger alerts for collection teams when bins approach capacity, while gas concentration limits can be modified in response to evolving safety standards or regulatory requirements. By externalizing threshold configuration, the system avoids hardcoded values and improves adaptability. This

allows operational parameters to be adjusted based on seasonal variations, geographic differences, or updated risk assessments.

3.4 Non-Functional Requirements

The system must integrate secure authentication and authorization mechanisms using Keycloak, which implements OAuth2 and OpenID Connect protocols. This ensures robust access control, secure session management, and role based permissions. Restricting access to sensitive data enhances trust, prevents unauthorized manipulation of system parameters, and ensures accountability through audit trails. Moreover, Keycloak's interoperability with external enterprise systems supports scalability and multi organizational deployment.

Data serialization and transmission must be optimized for minimal bandwidth consumption and low latency, even under unreliable network conditions. The MQTT based ingestion pipeline and Protobuf serialization should jointly ensure high throughput and efficient handling of large volumes of data generated by distributed IoT nodes.

The system architecture must be designed to support incremental scaling as the number of devices and data volume grows. Protobuf's strongly typed schema should facilitate long term maintainability by ensuring backward compatibility and simplifying future message schema evolution. Additionally, modular system components — including the ingestion pipeline, dashboard, and security services — should support seamless integration of new features and third party tools.

The web dashboard must offer an intuitive, user friendly interface, enabling users to configure operational thresholds, view data trends, and manage annotations without requiring in depth technical knowledge. This ensures that the system remains accessible to both technical and non-technical stakeholders.

A diagram in figure 3.3 to represent the how the components will interact with each others based on the previous key notes:

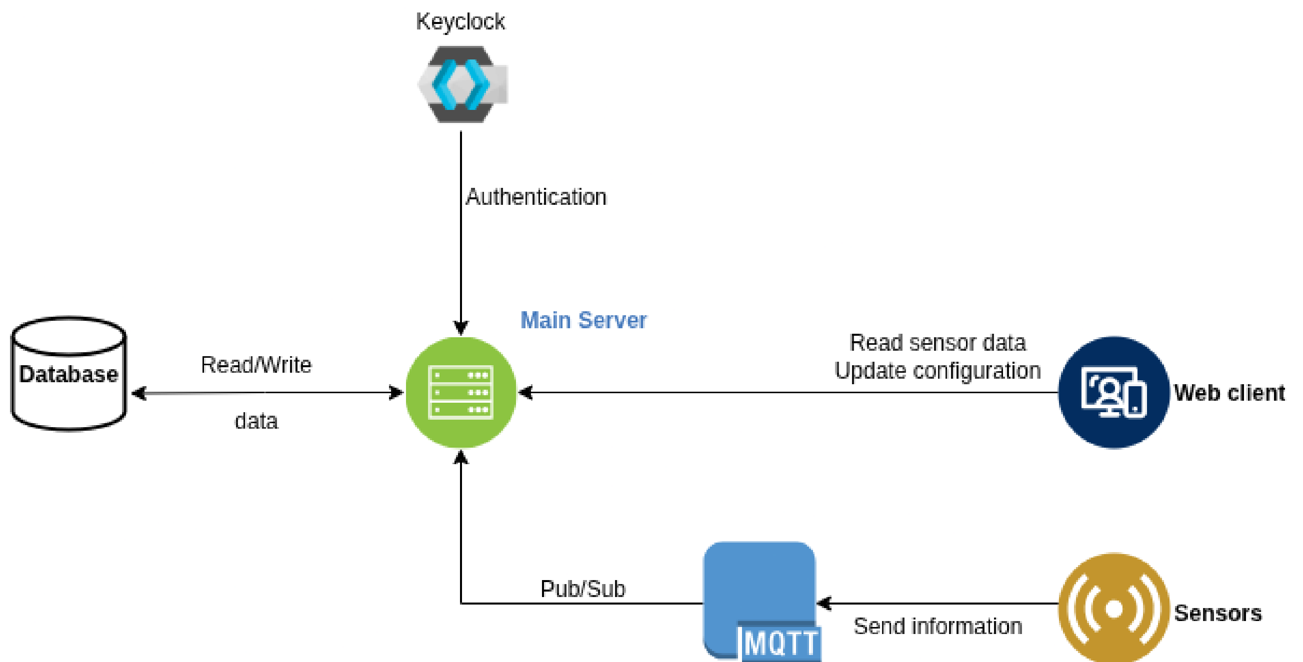

Figure 3.3: Detailed project diagram

3.5 Summary

Collectively, these functional and non-functional requirements form a resilient and scalable data lifecycle. This includes efficient encoding and transmission (Protobuf), reliable ingestion and structured storage (MQTT + database), enriched interpretation (annotations), actionable decision making (dashboard thresholds), and secure access (Keycloak). Together, these components illustrate how thoughtful system design leads to operational resilience, adaptability, and long term sustainability within smart waste management environments.

Chapter 4

Development

This chapter discusses about the process by which the Aggregator service, which is the backbone of a smart waste management platform, was made. The service has a clear REST and WebSocket Application Programming Interface (API) built with FastAPI, works with Keycloak for managing identities and access, gets telemetry from IoT sensors through MQTT and Long Range Wide Area Network (LoRaWAN), and stores and analyzes data using MongoDB. The goal of this chapter is to go into great detail about the architecture used, the development process and design decisions, all of the API endpoints, and the main packages and libraries used to build the system. The chapter ends with thoughts on how easy it is to maintain, how scalable it is, and what work needs to be done in the future.

4.1 Development Overview

4.1.1 Objectives

The main goals of this project are to create and put into use a safe, scalable, and analytics ready platform for smart waste management. These goals are in line with both the technical needs of modern IoT based systems and the real world needs of people who work in urban waste management. The system's main goals are to:

- **Provide secure, role based access to data and operations.** The system has a fine grained authentication and authorization system built into Keycloak that lets different types of users, like waste collectors, analysts, managers, administrators, and super administrators, use it. Each role gets access rights that match its duties. This keeps data safe, follows privacy rules, and stops unauthorized actions. This goal makes sure that the system can work in an environment with many stakeholders without putting security at risk.

- **Offer analytics ready endpoints and dataset windowing for downstream modeling and visualization.** The system has REST and WebSocket endpoints that show sensor readings, bin histories, and alert events to make it easier to make decisions based on data. The platform can do more than just store raw data; it can also create and annotate datasets within set time frames. This makes it possible to make training datasets for predictive modeling and testing. This makes sure that the infrastructure is not only working, but also helps with research and analytics tasks like predicting demand, finding anomalies, and optimizing routes.

- **Ensure observability and operational readiness.** Integrated observability features, such as health check endpoints that keep an eye on the database, MQTT broker, and authentication service, make the system more stable and easier to maintain. Swagger UI and ReDoc interfaces automatically create and make available complete API documentation. The system is also built with CI/CD pipelines in mind, which lets testing, deployment, and monitoring happen automatically to cut down on downtime and support ongoing improvement.

In summary, these goals together describe a system that is safe, open to new features, and ready for both real world use and future research. They stress the importance of finding a balance between technical rigor (through secure communication, structured data handling, and system observability) and practical impact.

4.1.2 Methodology

This project employs an API first and modular design technique, guaranteeing the system's extensibility, maintainability, and transparency for many stakeholders. The development approach establishes a solid foundation for real time operations and long term analytical applications through the use of well defined contracts and domain separation. The methodology prioritizes type safety, service decoupling, and automation, which together diminish complexity and enhance system reliability. The development workflow advanced in an iterative manner as outlined below:

1. **Define contracts via Pydantic schemas and document with OpenAPI/Swagger.** The system's data models and validation criteria were established with Pydantic, which ensures strict type and uniformity throughout all interactions. These schemas serve as definitive agreements between clients and the server, guaranteeing that inputs and outputs are consistent and well organized. The contracts are automatically incorporated into the resulting OpenAPI specification, yielding thorough and machine readable API documentation. Swagger UI and ReDoc interfaces were utilized to facilitate interactive exploration of available endpoints for developers, analysts, and operators.

2. **Implement routers per domain.** Each functional domain of the system was contained under a specific API router, addressing areas such as user management, bin monitoring, collection operations, analytics, alert management, and sensor dataset ingestion. This modular separation enables the software to expand without creating cross domain entanglement and facilitates a clear alignment of system features with stakeholder requirements. It facilitates more efficient testing, debugging, and future expansion, allowing for the introduction of additional domains with minimal disruption to existing ones.

3. **Integrate services as adapters.** External services, such as Keycloak for authentication, MongoDB for data persistence, MQTT for sensor data ingestion, and WebSocket for real time message streaming, were incorporated as adapters behind clearly specified interfaces. This design establishes rigid demarcations between application functionality and external dependencies, minimizing lock in and facilitating the independent replacement or scaling of services. The adapter pattern guarantees that external communication is abstracted, enhancing robustness and modularity within the system.

4. **Automate builds and deployments via GitLab CI/CD.** Continuous integration and deployment pipelines were established to automate the build, testing, and deployment procedures across development, staging, and production environments. GitLab CI/CD scripts manage tasks including dependency installation, test execution, containerization, and deployment to designated environments. This automation lowers manual labor, expedites feedback cycles, and guarantees uniform, replicable builds. Furthermore, it enhances the project's operational preparedness by allowing swift recovery from setbacks and promoting ongoing progress.

This methodology integrates optimal practices from contemporary software engineering with the practical limitations of IoT based systems. Emphasizing contract first design, modular routers, and adapter based service integration promotes system clarity and maintainability while ensuring the platform is resilient, scalable, and compatible with real world deployment scenarios.

4.1.3 Key Design Choices

The system's architecture and implementation are directed by intentional design decisions that harmonize performance, scalability, and maintainability. Each option demonstrates both technical appropriateness and conformity with the goals of secure, real time, and analytics capable waste management. The following delineates the most critical design decisions:

- **FastAPI for application logic.** The chosen web framework for the project is FastAPI, principally because of its inherent support for asynchronous I/O, facilitating the efficient management of concurrent requests and real time data streams. FastAPI autonomously produces OpenAPI specs and interactive documentation (via Swagger UI and ReDoc), guaranteeing that APIs are both discoverable and conducive to developer engagement. Moreover, its substantial dependence on Python type hints and Pydantic models enhances type safety, minimizes runtime mistakes, and augments the maintainability of the software.
- **Keycloak for identity and access management.** Authentication and permission are consolidated via Keycloak, an open source identity and access management system. The system implements secure, role based access management for various personas (e.g., collectors, analysts, administrators) by utilizing realm roles and token based authentication through the HTTP Bearer protocol. This solution guarantees adherence to contemporary security protocols while alleviating the responsibility of credential management within the application.
- **MongoDB for persistence.** MongoDB has been selected as the principal data repository due to its adaptability in managing semi structured sensor documents and its capacity to approximate time series storage. The schemaless architecture of MongoDB corresponds with the heterogeneity of IoT telemetry, while functionalities like secondary indices and aggregation pipelines facilitate efficient querying and analytics. Collections are structured by functional domains (bins, alerts, collectors, sensor readings, datasets), facilitating both real time operations and historical analysis.
- **MQTT for telemetry ingestion.** MQTT is utilized for ingesting and broadcasting telemetry from dumpster sensors through the `aiomqtt` client. MQTT is a lightweight protocol, built for constrained devices and adept at handling unpredictable network situations, rendering it ideal for IoT implementations. This delineation of responsibilities guarantees that each communication channel is utilized for

its specific advantages.

- **Pydantic Settings for configuration management.** Environment specific configurations (development, staging, production) are administered by Pydantic's settings module. This enables the injection of configuration values (such as database URIs, broker topics, or Keycloak endpoints) from environment variables in a type safe way. This method enhances repeatability across environments and mitigates the danger of misconfiguration.

- **Loose coupling through dependency modules.** To avert tight coupling between the web layer and integration layers, diminutive dependency modules are used as adapters for services including MongoDB, Keycloak, and MQTT. This solution enhances testability by allowing the injection of dummy clients during unit tests and promotes modularity by enabling the independent swapping or scaling of individual services. The notion of loose coupling underlies the system's extensibility and long term maintainability.

These design choices collectively establish a foundation that is both technically sound and flexible to future needs. They facilitate real time processing of sensor data, enforce secure access protocols, and maintain adaptability for both research and production environments.

4.2 System Architecture

4.2.1 Overview

The Aggregator service serves as the essential element of the proposed intelligent waste management platform. The architecture is designed as a web application that provides versioned APIs under the `/v1` prefix, adhering to an API first design philosophy. The service works as the integration hub between scattered IoT sensors, data storage systems, and user facing dashboards, facilitating both real time operations and long term analytical

capabilities. The architecture prioritizes modularity, scalability, and interoperability, allowing individual components to develop without compromising the integrity of the entire system. The primary responsibilities encompass the following:

- **Authenticate and authorize users via Keycloak.** The service entrusts identity and access management to a Keycloak instance, which facilitates token based authentication and role based authorization. This connection facilitates secure multi user access, granting permissions to various roles, including collectors, analysts, and administrators, in accordance with their operational duties. Externalizing identity management enhances security and streamlines user lifecycle administration.

- **Ingest and expose sensor data.** Information from sensors affixed to dumpsters (such as fill levels, temperature, humidity, and gas concentrations) is conveyed via MQTT. The service subscribes to telemetry streams, decodes the payloads (encoded in Protocol Buffers), and normalizes them prior to storing them in MongoDB. Upon ingestion, the data becomes accessible via HTTP endpoints that provide filtering by sensor identity and temporal intervals, thereby supporting both operational dashboards and subsequent analytical pipelines.

- **Manage domain entities and trigger alerts.** The service preserves domain specific entities, including bins, collectors, and their historical records, alongside sensor data. Modifications to these entities are verified against adjustable thresholds, and upon the detection of abnormalities or exceedances, alert events are created and recorded in the system. These notifications can subsequently be accessed using REST APIs or streamed in real time via WebSockets, enabling operational teams to react swiftly to urgent situations such as bins approaching capacity or dangerous gas emissions.

- **Provide analytics and dataset endpoints.** The service provides analytics ready endpoints enabling users to query historical data within adjustable date ranges, obtain time series data for bins and collectors, and create datasets for study and

predictive modeling. These features connect operational monitoring with strategic analysis, allowing the platform to function as both a management tool and a research supporting infrastructure.

The Aggregator service integrates many functionalities—secure access control, telemetry ingestion, entity management, alerting, and analytics—into a cohesive design. This stratified design guarantees that the system facilitates routine waste management operations while also establishing a foundation for sophisticated data driven decision making inside smart city programs.

4.2.2 Web Layer

The web layer serves as the external interface of the Aggregator service, presenting the system's capabilities to both human users and machine clients via well defined HTTP endpoints. It is constructed with modularity and separation of concerns, facilitating maintainability, extensibility, and distinct domain boundaries. The implementation adheres to the standards of FastAPI and Pydantic, guaranteeing robust typing, automatic validation, and self documenting APIs. The primary components are organized as follows:

- **`app/web/api/router.py`.** This module functions as the primary access point for the web API. It organizes the routers for each functional domain (e.g., users, bins, collectors, analytics, alerts, dataset) under the `/v1` prefix. This versioning technique ensures backward compatibility as the system progresses, guaranteeing that current customers are not impacted by subsequent modifications or enhancements.

- **Domain specific API modules.** Each domain is executed as an autonomous module situated within `app/web/api/<domain>`. Each module has a `schema.py` file that delineates the Pydantic models utilized for request validation and response serialization, alongside a `views.py` file that supplies the associated route handlers. This separation enables schemas to develop independently of business logic, enhances reusability across endpoints, and ensures consistency in data representation throughout the system.

- **Static documentation and interactive endpoints.** Alongside programmatically created OpenAPI/Swagger specifications, static documentation assets are preserved under `app/web/static/docs`. Router based documentation endpoints are made available to furnish developers and operators with easy access to usage instructions, API references, and troubleshooting guides. This method not only streamlines onboarding and minimizes integration challenges but also guarantees adherence to the project's goals of observability and operational preparedness.

The web layer simplifies the system's internal complexity through a consistent and well structured API boundary. Utilizing Pydantic for schema enforcement and FastAPI for automatic documentation creation guarantees both robustness and transparency. Furthermore, the modular architecture facilitates the gradual enhancement of functionality, permitting the integration of supplementary domains or endpoints without interfering with current operations. This architectural choice corresponds with the overarching objective of establishing a durable and scalable platform capable of supporting future research and operational requirements.

4.2.3 Identity and Access Management

An essential component of the system's architecture is the implementation of secure, role based identity and access management. This guarantees that critical processes, including sensor data management and administrative settings, are accessible solely to authorized individuals based on their responsibilities. The implementation utilizes Keycloak, a recognized open source identity and access management system, which offers support for OAuth2/OpenID Connect and centralized user lifecycle management. The integration is organized in the following manner:

- **`services/keycloak` module.** This component encompasses all interactions with the Keycloak server. It supplies application level dependencies to obtain and verify bearer tokens produced by Keycloak, guaranteeing that each API call is authenticated prior to accessing system resources. The module provides methods for

interacting with Keycloak's administrative endpoints, including user creation, role assignment, and credential management. By segregating this functionality, the system preserves loose coupling between business logic and identity management, while maintaining the adaptability to modify or substitute the IAM provider as necessary.

- **Role based access control (RBAC).** The platform delineates a series of roles corresponding to the operational personas inside the waste management workflow: `waste_collector`, `analyst`, `manager`, `admin`, and `super_admin`. Each position is associated with a certain set of rights, encompassing simple data gathering duties to advanced system administration. A waste collector may solely submit sensor reports and access allocated bins, whereas an analyst can query past data and create datasets, and administrators oversee user accounts and thresholds. The existence of a `super_admin` job facilitates meta level governance, including the establishment of new organizations and the management of multi tenant installations.

- **Users API.** The Users API offers endpoints for authentication, profile access, role specific registration, and modifications to user profiles and credentials. Authentication is token based, guaranteeing that sensitive actions are accessible solely to logged in users with appropriate role assignments. The registration processes include role scoping, so minimizing privilege escalation by limiting the roles that can be self assigned. Moreover, profile and password management endpoints enable users to manage their account information, thereby decreasing administrative burden while ensuring security.

This architecture ensures that access to data and operations is strictly regulated by role based policies, hence reducing the danger of unauthorized acts. The dependence on Keycloak guarantees standards compliant authentication and facilitates interaction with external identity providers (e.g., LDAP, SAML, or social logins), thereby positioning the platform for use in larger organizational settings. This method precisely corresponds with the project's aim of facilitating secure, multi user access to IoT based waste management systems.

4.2.4 Data Ingestion and Storage

The acquisition and retention of sensor data are the foundation of the system's functionality, facilitating operational oversight and long term analysis. The architecture prioritizes scalability, adaptability, and compatibility with many IoT communication standards. The architecture delineates responsibilities among message delivery, payload decoding, and database access, thus guaranteeing distinct boundaries and facilitating maintenance. The primary components are delineated as follows:

- **`services/mqtt`.** This module oversees the MQTT client's lifespan, encompassing connection, subscription, and orderly disconnection. It additionally offers auxiliary tools for disseminating telemetry and obtaining retained messages when required. By encapsulating MQTT functionality within a specialized service, the solution segregates message transmission from application logic, facilitating comprehensive error management, reconnection protocols, and effective message parsing. This is essential as dumpster sensors function under diverse network settings and must consistently provide fill levels, gas concentrations, and environmental variables.

- **`services/lorawan`.** The initial implementation primarily emphasizes MQTT, however the system also plans for integration with LoRaWAN devices, prevalent in low power wide area networks (LPWANs). The `services/lorawan` module offers tools for decoding LoRaWAN payloads and standardizing them into the internal schema. This forward compatible design guarantees that the architecture may support several IoT connectivity standards without necessitating substantial modifications to upstream components.

- **`services/mongo`.** The persistence layer is contained within the `services/mongo` module, which encapsulates the asynchronous PyMongo client and incorporates lifecycle hooks for dependency injection into FastAPI routes. This solution guarantees the secure management of database sessions across requests while reducing boilerplate code in the higher level application logic. The module offers a cohesive

interface to collections, ensuring consistency in data access patterns and facilitating the extension or replacement of the underlying database technology in subsequent generations.

- **Dataset definition and analytics readiness.** In addition to raw data ingestion, the system facilitates the specification of datasets via temporal slicing. Readings may be categorized by sensor identifier, label, or date range, thus creating datasets that are readily applicable for subsequent analytical tasks such as anomaly detection, forecasting, or supervised machine learning. This functionality corresponds with the overarching project objective of facilitating research and decision making by connecting operational telemetry with organized analytical tools.

The system integrates protocol aware ingestion modules with a versatile persistence layer to provide reliable capture, normalization, and storage of sensor data for both short term monitoring and long term analysis. The modular design enables interoperability with various IoT protocols and offers a scalable basis for future improvements, including real time stream processing and interaction with data lakes for big data analytics.

4.2.5 Messaging and Events

The system supports asynchronous processes using a specialized messaging backbone, in addition to synchronous request–response exchanges provided by the HTTP API.

- **Alerts and event extension.** Currently, alert production and acknowledgment are managed synchronously via the API and database operations. The design plans to include these methods in an event driven approach, allowing threshold breaches to be broadcast through WebSocket connections, facilitating near real time notice streaming to numerous downstream users. This would enable seamless integration with external systems (e.g., dashboards, mobile push notification services, or predictive analytics pipelines) without exerting additional strain on the core API.

The eventing layer signifies a logical progression for the system, enabling its transformation from a predominantly synchronous web service to a hybrid platform that integrates API driven and event driven paradigms. This extensibility is essential in IoT dense contexts, where scalability and responsiveness are crucial.

4.2.6 Configuration and Environments

The project employs a configuration method that emphasizes reproducibility, maintainability, and environment specific isolation. All environment variables and secrets are centrally established and controlled using Pydantic settings, guaranteeing type safety and validation during initialization.

- **`app/settings`.** The `app/settings` module offers a consolidated access point for setup across many environments. It extends the `pydantic settings` library to interpret environment variables and validate them according to rigorously defined schemas. Environment specific configurations (development, staging, and production) are facilitated through overrides, ensuring uniform behavior across deployments while permitting context specific customisation (e.g., distinct database URIs, API keys, or broker endpoints).
- **Continuous Integration and Deployment (CI/CD).** To maintain operational consistency, CI/CD pipelines are delineated in `.gitlab-ci*.yml`. These pipelines automate the construction, testing, and deployment processes for each environment, minimizing the likelihood of manual errors and ensuring consistent releases. The deployment method guarantees that features are tested in isolation on staging prior to their promotion to production, while development builds facilitate swift feedback loops for contributors.

The integration of configuration management with CI/CD pipelines creates a basis for operational excellence. They empower developers and operators to effectively manage various environments, enforce configuration consistency, and provide dependable software upgrades in accordance with contemporary DevOps methodologies.

4.3 Endpoints

Table 4.1 details the system's RESTful API endpoints. Each item lists the HTTP method, endpoint path, and some description of its function. The API's logical layout divides the table into themed areas for global documentation, user management, MQTT communication, analytics, collectors, alarms, and bin management.

The Global Documentation area provides access to Swagger and ReDoc generated API documentation, as well as a health check endpoint. The Users component includes authentication, registration, profile management, and administrative operations endpoints. MQTT endpoints provide message publication, sensor data retrieval, and analytics dataset window management.

The **Analytics** has read only endpoints for accessing time series data and collection histories, whereas the **Collectors** portion enables full CRUD actions on collectors, including collection event triggering. The **Alerts** section covers threshold management, alert acknowledgment, and real time WebSocket notifications. The Bins area provides endpoints for managing smart garbage bins, monitoring their status, and accessing live data streams via WebSocket connections.

Method	Endpoint	Description
	Global Documentation	
GET	`/docs`	Swagger UI
GET	`/redoc`	ReDoc UI
GET	`/openapi.json`	OpenAPI schema
GET	`/v1/health`	Service health check
GET	`/v1/docs`	Swagger UI (router)
GET	`/v1/swagger-redirect`	OAuth2 redirect
GET	`/v1/redoc`	ReDoc UI (router)
	Users (/v1/users)	
POST	`/login`	Obtain Keycloak tokens
GET	`/profile`	Current user profile
POST	`/super-admin/init`	Initialize `super_admin`
POST	`/register/{role}`	Register with role
GET	`/`	List users (admin scopes apply)
DELETE	`/{user_id}`	Delete user
PATCH	`/{user_id}`	Update profile fields
PATCH	`/password`	Change current user's password
	MQTT (/v1/mqtt)	
POST	`/`	Publish MQTT message
GET	`/sensor`	Query sensor readings
POST	`/dataset`	Create dataset time window
GET	`/dataset/{dataset_id}`	Readings for dataset window
GET	`/dataset/label/{label}`	Readings by label
GET	`/dataset/date-range`	Readings by explicit date range
	Analytics (/v1/analytics)	
GET	`/bin/{bin_id}/history`	Time-series by metric
GET	`/collector/{collector_id}/history`	Collection history

Collectors (/v1/collectors)		
GET	/	List collectors
GET	/{collector_id}	Get collector
POST	/	Create collector
PUT	/{collector_id}	Update collector
POST	/{collector_id}/collect	Record bin collection
DELETE	/{collector_id}	Delete collector
Alerts (/v1/alerts)		
GET	/active	List active alerts
POST	/thresholds	Create/update thresholds
GET	/thresholds/{bin_id}	Get thresholds for a bin
WS	/ws/thresholds	WebSocket for threshold-triggered alerts
PUT	/acknowledge/{alert_id}	Acknowledge alert
Bins (/v1/bins)		
GET	/status	List current status of all bins
GET	/{bin_id}	Get a bin by ID
POST	/	Create a bin
PUT	/{bin_id}	Update a bin
DELETE	/{bin_id}	Delete a bin
WS	/ws	WebSocket for live updates

Table 4.1: API Endpoints Reference

4.4 Packages and Dependencies

The following core packages are used (see `requirements.txt`):

- **fastapi**: Web framework for building the API and WebSocket endpoints.
- **hypercorn**: Asynchronous Server Gateway Interface (ASGI) server for running the application.

- **loguru**: Structured logging with minimal configuration.
- **ujson**: High performance JSON serialization.
- **pydantic**: Data validation and settings modeling for request/response schemas.
- **pydantic-settings**: Environment driven configuration management.
- **aiomqtt**: Async MQTT client for publishing/subscribing to telemetry topics.
- **pymongo**: MongoDB driver used via async wrappers for data persistence.
- **passlib[bcrypt]**: Password hashing utilities where needed.
- **python-multipart**: Form data parsing, supporting file uploads and multi-part forms.
- **email-validator**: Email address validation in user related schemas.
- **requests**: HTTP client used for calling Keycloak Admin REST endpoints.
- **yarl**: URL parsing utilities.
- **protobuf**: Protocol Buffers support for structured sensor payloads (see `app/protos`).
- **pycryptodome**: Cryptographic primitives potentially used for secure payload handling.

4.5 Notable Implementation Details

The system integrates several critical mechanisms to enhance security, data accessibility, and real time responsiveness. User management is handled through self service API endpoints (`PATCH /v1/users/me` and `PATCH /v1/users/password`), allowing authenticated users to update their profiles and credentials securely. Role based authorization is enforced via Keycloak realm roles and Bearer token claims, while administrative operations

are executed through the Keycloak Admin API with rigorous header validation and error handling. To mitigate security risks, field level whitelisting restricts updates to safe attributes such as email, first name, last name, username, and, for administrators, the account enable flag.

For experimental and analytical use cases, the dataset subsystem enables flexible temporal slicing of sensor data. Endpoints support dataset creation and retrieval based on specific time windows, unique identifiers, labels, or explicit date ranges. This capability ensures reproducible selection of temporal data subsets, facilitating consistent benchmarking and comparative evaluations across experiments.

To support live monitoring and system awareness, alert thresholds can be configured globally, per tag, or per bin. Upon threshold breaches, events are broadcast through a dedicated WebSocket channel to connected clients, enabling near real time updates in dashboards and other external systems. This event driven communication model enhances responsiveness and situational awareness without imposing additional computational overhead on the core API.

4.6 Summary

This chapter went into great detail about how the Aggregator service was built and how its system architecture works. It talked about the web API, identity integration, data ingestion, storage, and messaging layers. It made a list of all the available endpoints and talked about the most important Python packages that helped with the implementation. The architecture focuses on modularity, security, and extensibility, which means that the platform can grow with new analytics, event processing, and device integrations.

Chapter 5

Tests and Discussion

5.1 Swagger

In contemporary software development, especially in the design and execution of RESTful APIs, precise documentation and consistent communication are essential for efficiency, interoperability, and maintainability. Swagger, now integrated into the OpenAPI ecosystem, has become a prevalent framework that fulfills these requirements. Swagger facilitates developers in designing, visualizing, and testing endpoints effortlessly by offering a standardized, machine-readable specification for APIs. Its importance beyond simple documentation; it enables automatic code creation, guarantees consistency between implementation and documentation, and improves collaboration among development teams.

A summary of the API endpoints generated by the Swagger UI documentation tool displayed in figure 5.1. The endpoints are organized by resource tags, such as `users` and `bins`. Each entry specifies the HTTP method (e.g., `GET`, `POST`), the URL path, and a brief functional description.

Figure 5.1: High-level overview of the API endpoints

An illustration in figure 5.2 of the data schemas for several Data Transfer Objects (DTOs) used in the API. The figure details the structure of the `UserSelfUpdateRequest`, `UserUpdateRequest`, and `ValidationError` models. For each model, the fields, their corresponding data types (e.g., `string`, `null`), and validation constraints, such as character limits (e.g., username must be between 3 and 50 characters), are defined.

Figure 5.2: Data schemas for various API requestss

Detailed documentation in figure 5.3 for the user authentication endpoint, `POST /v1/users/login`. The interface shows an example of the required JSON request body, containing the `username` and `password` fields. It also documents the structure of a successful `200 OK` response, which returns a JSON object with an `access_token`, `refresh_token`, and other session data.

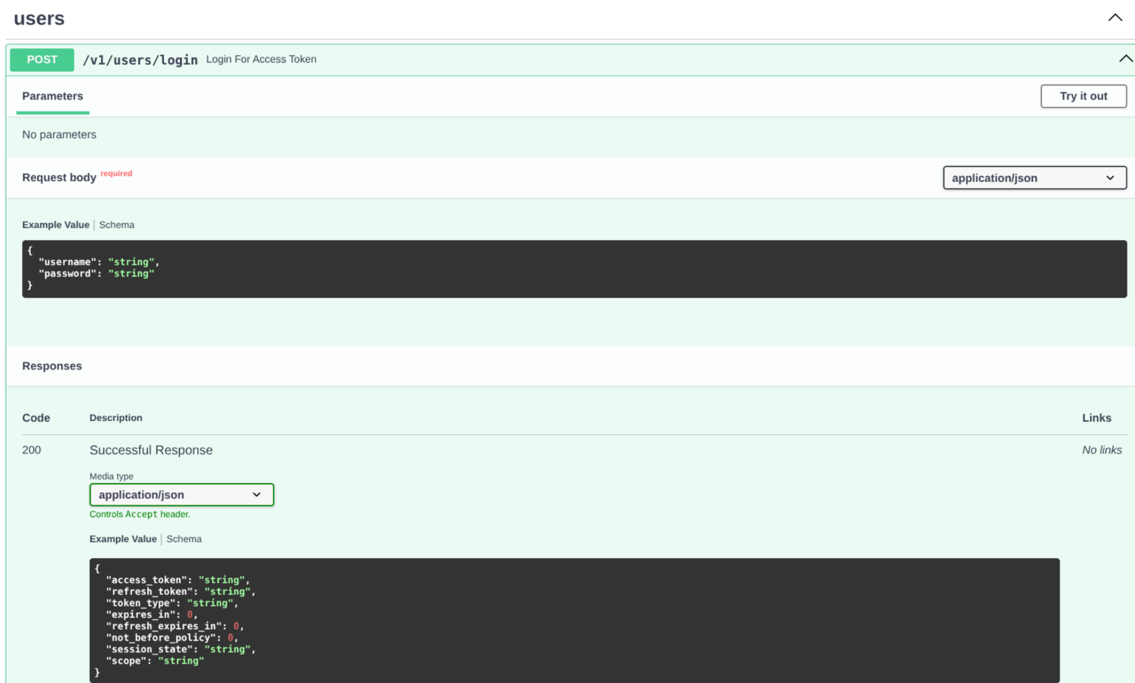

Figure 5.3: Detailed view of the user login endpoint

5.2 Benchmarking

5.2.1 Introduction

Benchmarking is essential for the systematic assessment of system performance, allowing academics and practitioners to evaluate efficiency, scalability, and dependability in controlled environments. Benchmarking offers objective proof of performance strengths and limitations by submitting a system to specified workloads and comparing its behavior across various configurations. This procedure is crucial for confirming design decisions,

finding bottlenecks, directing optimization initiatives, and facilitating equitable comparisons with different methodologies or related research.

In concurrent systems and server architectures, benchmarking is crucial, as differences in worker allocation, request management, and resource utilization can significantly affect end-to-end response times. Rigorous benchmarking enables the quantification of parameter impacts and facilitates significant conclusions about system scalability and operational resilience.

5.2.2 Benchmark test

To evaluate the performance of the system under concurrent workloads, a benchmark test was conducted using a predefined API endpoint. The objective was to measure response latency, stability under parallel requests, and the consistency of returned data. The following subsections present the benchmark procedure, including the structure of the response payload, the request execution logic, and the parallel dispatching mechanism, the data is a samples fetched from the bme688 development kit, stored in the mongodb.

The benchmark targets an endpoint returning sensor data in JSON format. Understanding the structure of the response is essential to verifying payload integrity and assessing the server's ability to handle data serialization under load.


```json
{
        "_id": "",
        "label": "ash",
        "data_points": [
        {
                "data_point_id": 20851,
                "resistance_gassensor": 9563.658203,
                "temperature": 36.008518,
                "pressure": 932.915039,
```

```
            "relative_humidity": 19.490475,
            "time_since_poweron": 4205921,
            "real_time_clock": 1688776216,
            "error_code": 0,
            "cycle_step_index": 0,
            "cycle_id": 2086
        },
        ...
}
```


This response illustrates a typical sensor dataset consisting of environmental readings and device metadata, which serves as the basis for evaluating both throughput and consistency during high load conditions.

The following code fragment is responsible for issuing HTTP requests to the API endpoint, recording the response times, and categorizing responses based on their HTTP status codes. This enables the calculation of minimum, maximum, and average latencies, which are later used for statistical analysis.

```
url = "http://0.0.0.0:9999/api/v1/samples/bme688?limit=200&offset=0"


def make_request():
        global count200, count400, count_other, min_time, max_time


        start_time = time.time()
        response = requests.request("GET", url, headers=headers, data=payload)
        elapsed_time = time.time() - start_time
        request_times.append(elapsed_time)
        # Update min and max times
        min_time = min(min_time, elapsed_time)
        max_time = max(max_time, elapsed_time)
```

```
if response.status_code == 200:
count200 += 1
elif response.status_code == 400:
count400 += 1
print(response.text)
else:
print(response.text)
count_other += 1
```

This method provides the core measurement functionality, enabling precise timing of individual requests and capturing potential anomalies in server behavior through non-200 responses.

To simulate realistic high load conditions, multiple threads are employed to issue concurrent API requests. This approach allows the benchmark to assess the scalability of the server under simultaneous client interactions.

```
request_start = time.perf_counter()
with ThreadPoolExecutor(max_workers=100) as executor:
        futures = [executor.submit(make_request, ) for i in range(1000)]
        for future in as_completed(futures):
                response = future.result()
request_end = time.perf_counter()
```

By leveraging a *ThreadPoolExecutor*, the system is stressed with 1,000 parallel requests, enabling the extraction of performance metrics such as total execution time and system responsiveness under concurrency.

5.2.3 Results

The following abbreviations are used throughout the analysis to denote key performance metrics related to request handling efficiency:

- **W-Count**: Worker Count
- **T-R Time**: Total Request Time
- **MIN-R Time**: Minimum Request Time
- **MAX-R Time**: Maximum Request Time
- **AVG-R Time**: Average Request Time

The following table presents comprehensive performance outcomes from three distinct trials for each worker configuration (W-Count = 1, 5, 10, 20). It encompasses total request time, as well as minimum, maximum, and average response times, in addition to success and failure counts.

The results unequivocally indicate a robust inverse correlation between the number of workers and total execution time. As the workforce expands, both Total Request Time (T-R Time) and Average Request Time (AVG-R Time) significantly diminish. For instance, with one worker, the entire request time is roughly 28 to 29 seconds, however with twenty workers, it reduces to about 4.2 seconds. This underscores the significant scalability advantage attained via parallelism.

Furthermore, augmenting the workforce diminishes the fluctuation between the minimum and maximum response times, signifying enhanced stability and consistency in request handling capabilities. Throughout all trials, there are no failures, proving that enhancements in performance do not undermine dependability. The discrepancies across runs are negligible, indicating consistent and reliable performance across all configurations.

Table 5.1: Hypercorn workers benchmarking

W-Count	Run	T-R Time	MIN-R Time	MAX-R Time	AVG-R Time	Success	Fail
1	1	29.262	0.408	3.393	2.898	1000	0
1	2	27.798	0.237	4.489	2.748	1000	0
1	3	27.469	0.501	3.859	2.713	1000	0
5	1	6.871	0.100	1.834	0.665	1000	0
5	2	6.478	0.092	1.194	0.615	1000	0
5	3	6.333	0.058	1.388	0.605	1000	0
10	1	4.904	0.054	1.105	0.474	1000	0
10	2	4.709	0.068	1.090	0.451	1000	0
10	3	4.698	0.042	1.074	0.453	1000	0
20	1	4.172	0.061	0.992	0.398	1000	0
20	2	4.205	0.053	1.054	0.402	1000	0
20	3	4.261	0.058	1.050	0.410	1000	0

This table displays the mean performance values derived from three iterations for each worker count indicated in Table 5.1. Consolidating the data eliminates run-to-run variability and provides a clear, high level comparison of worker setups.

The brief summary facilitates a more straightforward assessment of the performance enhancement achieved by scaling from 1 to 20 worker. Readers can readily discern the pattern without requiring interpretation of individual experimental trials. It is particularly beneficial for debates, conclusions, and graphic representations in the analysis portion.

Table 5.2: Simplified benchmark overview

W-Count	T-R Time	MIN-R Time	MAX-R Time	AVG-R Time
1	28.176	0.382	3.914	2.786
5	6.561	0.083	1.472	0.628
10	4.770	0.055	1.090	0.459
20	4.213	0.057	1.032	0.403

The subsequent images visually depict the correlation between worker concurrency and system performance, based on the metrics given in Tables 5.1 and 5.2. The graphs illustrate the total, minimum, maximum, and average request durations across various worker configurations, enabling a more precise comparison of performance trends and emphasizing the impact of scalability on response behavior. This graphic representation facilitates the identification of proportional enhancements, stability trends, and possible declining benefits as the workforce expands.

Total request time to complete the test workload, plotted against the number of worker processes (W). The overall execution time decreases dramatically as more workers are added, with diminishing returns observed after $W = 8$.

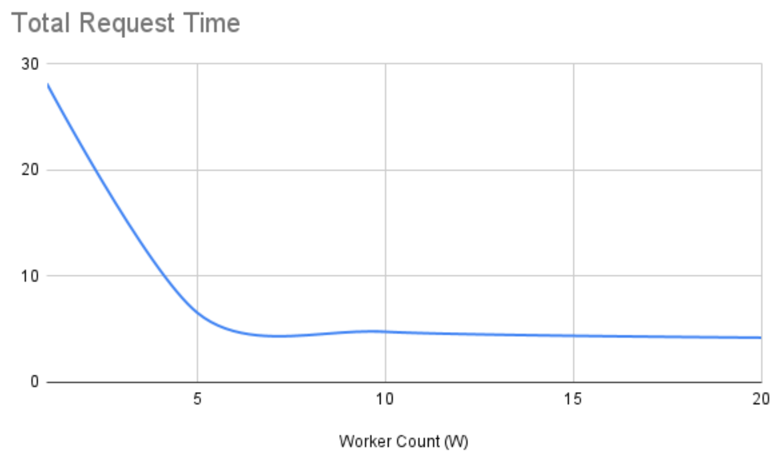

Figure 5.4: Total request time per #workers

Minimum request time as a function of the number of worker processes (W). The graph illustrates that even the fastest requests benefit from an increased worker count, with the minimum time stabilizing at a low value of approximately 0.05 seconds for $W \geq 8$.

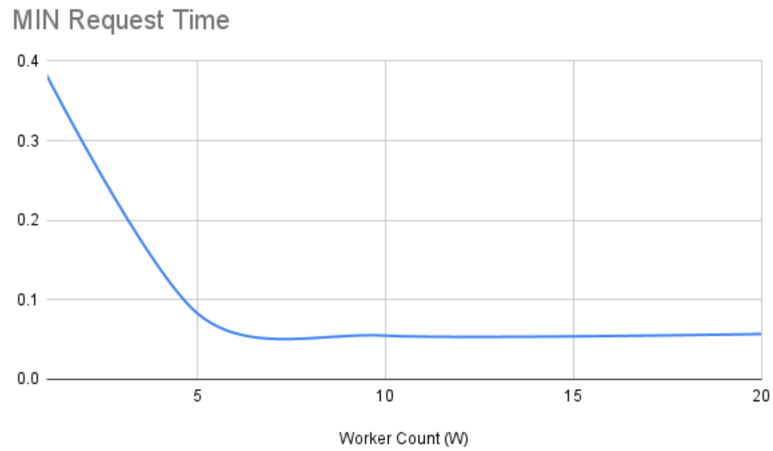

Figure 5.5: Minimum request time per #workers

Maximum request time as a function of the number of worker processes (W). The graph shows a sharp decrease in the maximum request time as the worker count increases from 1 to approximately 8, after which the performance plateaus around 1 second.

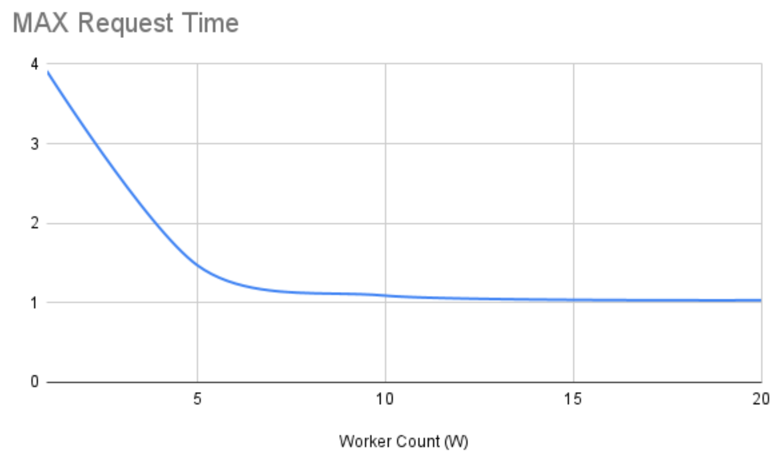

Figure 5.6: Maximum request time per #workers

Average request time relative to the number of worker processes (W). A significant performance improvement is observed up to $W = 8$, with the average request time stabilizing at approximately 0.4 seconds for higher worker counts.

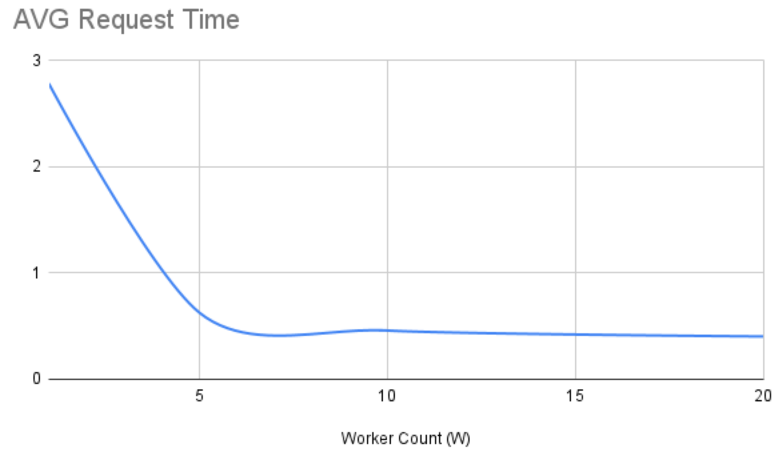

Figure 5.7: Average request time per #workers

Chapter 6

Conclusions

This project developed and executed a scalable, real time backend system to collect, process, and visualize data from IoT enabled dumpsters to address urban waste management issues and ensure worker safety, and facilitate data driven decision making in municipalities.

The research indicates that smart city technology and enough backend infrastructure can provide proactive waste management. Real time monitoring to mitigate environmental pollution, public health risks, and operational inefficiencies.

This study corroborates prior studies on IoT enabled urban infrastructure and intelligent waste management systems, illustrating that sensor based technology can enhance municipal safety and efficiency. Many research focus exclusively on hardware or low frequency data collecting, but this work shows end-to-end backend integration, high throughput performance, low latency processing, and system robustness under simulated city scale workloads.

This report presents a comprehensive architecture for the upgrading of backend waste management through IoT integration. The results underscore the imperative for real time monitoring, predictive analytics, and interactive visualization to enhance safety, efficiency, and environmental sustainability in urban operations.

Bibliography

[1] Anam, K., Rofi, D. N., & Meiyanti, R. (2023). Monitoring System for Temperature and Humidity Sensors in the Production Room Using Node-Red as the Backend and Grafana as the Frontend. In Journal of Systems Engineering and Information Technology (JOSEIT) (Vol. 2, Issue 2, pp. 68–76). Ikatan Ahli Informatika Indonesia (IAII). https://doi.org/10.29207/joseit.v2i2.5222

[2] P. Sangat, M. Indrawan-Santiago, and D. Taniar, “Sensor data management in the cloud: Data storage, data ingestion, and data retrieval," Concurrency and Computation: Practice and Experience, vol. 30, no. 1. Wiley, Nov. 02, 2017. doi: 10.1002/cpe.4354. Available: https://doi.org/10.1002/cpe.4354

[3] Y. Huang, J. Fan, Z. Yan, S. Li, and Y. Wang, “A Gas Concentration Prediction Method Driven by a Spark Streaming Framework," Energies, vol. 15, no. 15. MDPI AG, p. 5335, Jul. 22, 2022. doi: 10.3390/en15155335. Available: https://doi.org/10.3390/en15155335

[4] L. T. De Paolis, V. De Luca, and R. Paiano, “Sensor data collection and analytics with thingsboard and spark streaming," IEEE, Jun. 2018. doi: 10.1109/eesms.2018.8405822. Available: https://doi.org/10.1109/eesms.2018.8405822

[5] L.-M. Ang, K. P. Seng, A. M. Zungeru, and G. K. Ijemaru, “Big Sensor Data Systems for Smart Cities," IEEE Internet of Things Journal, vol. 4, no. 5. Institute of Electrical and Electronics Engineers (IEEE), pp. 1259–1271, Oct. 2017. doi: 10.1109/jiot.2017.2695535. Available: https://doi.org/10.1109/jiot.2017.2695535

[6] M. B. Krassovski, G. E. Lyon, J. S. Riggs, and P. J. Hanson, “Near-real-time environmental monitoring and large-volume data collection over slow communication links," Geoscientific Instrumentation, Methods and Data Systems, vol. 7, no. 4. Copernicus GmbH, pp. 289–295, Oct. 25, 2018. doi: 10.5194/gi-7-289-2018. Available: https://doi.org/10.5194/gi-7-289-2018

[7] Y. Huang, S. Li, J. Fan, Z. Yan, and C. Li, “A Spark Streaming-Based Early Warning Model for Gas Concentration Prediction," Processes, vol. 11, no. 1. MDPI AG, p. 220, Jan. 10, 2023. doi: 10.3390/pr11010220. Available: https://doi.org/10.3390/pr11010220

[8] R. Hummen, M. Henze, D. Catrein, and K. Wehrle, “A Cloud design for user-controlled storage and processing of sensor data," IEEE, Dec. 2012. doi: 10.1109/cloudcom.2012.6427523. Available: https://doi.org/10.1109/cloudcom.2012.6427523

[9] Abbott, M. L.; Fisher, M. T. (2015). *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise (2nd ed.)*. Pearson. ISBN 978-0-13-403280-1.

[10] Richardson, C. (2018). *Microservices Patterns: With examples in Java*. Manning. ISBN 978-1-63835-632-5.

[11] Pacheco, V. F. (2018). *Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices*. Packt Publishing. ISBN 978-1-78847-403-0.

[12] Schlossnagle, T. (2007). *Scalable Internet Architectures*. Sams Developer’s Library. ISBN 978-0-672-32699-8.

[13] Gurung, N.; Shrestha, S.; Chulyadyo, R. (2025). “Scalability in Microservices: A systematic literature review", *Journal of Computer Science and Technology*, Vol. 25, No. 2, e11. DOI:10.24215/16666038.25.e11.

[14] Shah, R.; Jagtap, S.; Jain, V. (2025). “Architecting Analytics-Driven Mobile Ecosystems: Scalable Backend Frameworks for Intelligent Data Flow and Real-Time User Insights", *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, Vol. 6, No. 2, 83-91. DOI:10.63282/3050-9262.IJAIDSML-V6I2P109.

[15] Krishnan, A. (2025). “Backend Development Optimizations for Enhanced Operational Efficiency: A Literature Review", *International Journal of Research in Modern Engineering & Emerging Technology (IJRMEET)*, Vol. 10, No. 11. DOI:10.63345/ijrmeet.org.v10.i11.6.