UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU EN ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Higiea: Design and Implementation of a Waste Management Infrastructure for Smart Cities

---

Autor: Jan Manel Morales Bes

Director:	**Dr. Eloi Puertas i Prats**
Realitzat a:	**Departament de Matemàtiques i Informàtica**

Barcelona, **15 de gener de 2025**

Abstract

This bachelor's thesis investigates how real-time data and smart technologies can transform traditional municipal waste collection into a more sustainable and efficient process. Drawing on concepts learned throughout four years of computer engineering studies—ranging from software design and distributed systems to software project methodologies—this project introduces *Higiea*, a platform designed to integrate key ideas of real-time monitoring and automated decision-making. By focusing on practical concerns such as on-demand route generation, dynamic container tracking, and resource optimization, it aims to reduce both environmental impact and operational costs.

The work also explores how smarter, more adaptive waste management can serve as a building block in broader "smart city" visions, aligning with urban sustainability efforts and citizen well-being. Readers will discover how the platform leverages modern software development practices, how data-driven insights can streamline daily operations, and how these principles can be extended or adapted for larger-scale implementations.

Ultimately, this thesis not only addresses the pressing issue of inefficient waste collection but also demonstrates how academic theory can be transformed into a functional, real-world prototype. From software engineering fundamentals to real-time data handling and city-scale challenges, the project showcases how a well-designed technical solution can address immediate community needs while providing a flexible foundation for future enhancements.

Resum

Aquest treball de final de grau explora com l'ús de dades en temps real i tecnologies intel·ligents pot modernitzar la recollida de residus municipals, fent-la més sostenible i eficient. Aprofitant els coneixements adquirits durant quatre anys d'estudis d'enginyeria informàtica —des del disseny de programari fins a la gestió de projectes—, es presenta *Higiea*: una plataforma que permet una gestió dinàmica dels contenidors i la planificació de rutes sota demanda. L'objectiu principal és reduir tant l'impacte mediambiental com els costos operatius, tot garantint un servei de qualitat als ciutadans.

En aquest sentit, la proposta s'emmarca dins de la idea de "ciutats intel·ligents", on l'aplicació de solucions tecnològiques avançades pot millorar l'eficiència urbana i el benestar general. El lector descobrirà com la plataforma utilitza metodologies de desenvolupament de programari, sistemes distribuïts i anàlisi de dades per oferir un sistema robust i flexible que es pot adaptar a diverses dimensions de ciutat.

L'estudi no només aborda un problema municipal concret —l'eficiència en la recollida de residus—, sinó que també demostra la transferència de coneixements acadèmics cap a una solució de programari completa i aplicable a diferents escenaris reals. Des dels fonaments de l'enginyeria del programari fins al tractament de dades en temps real i els desafiaments a escala urbana, el projecte demostra com una solució tècnica ben dissenyada pot donar resposta a necessitats immediates de la comunitat, alhora que estableix una base flexible per a millores futures.

Resumen

Este proyecto de fin de grado analiza cómo la aplicación de datos en tiempo real y tecnologías inteligentes puede modernizar la recogida de residuos en entornos municipales, haciéndola más sostenible y eficiente. A partir de los conocimientos adquiridos durante cuatro años de estudios en ingeniería informática —desde el diseño de software hasta la gestión de proyectos—, se presenta *Higiea*: una plataforma que facilita una gestión dinámica de contenedores y una planificación de rutas bajo demanda. El principal objetivo es reducir tanto el impacto medioambiental como los costes operativos, ofreciendo a la vez un servicio de mayor calidad a la ciudadanía.

La propuesta se enmarca en la visión de "ciudades inteligentes", donde la incorporación de soluciones tecnológicas avanzadas mejora la eficiencia y el bienestar urbano. El lector encontrará detalles sobre cómo la plataforma integra metodologías de desarrollo de software, sistemas distribuidos y análisis de datos para ofrecer un sistema sólido y versátil, apto para diferentes escalas de implementación.

El estudio no solo aborda un problema específico —la eficacia en la recolección de residuos—, sino que también demuestra cómo el conocimiento académico puede transformarse en una solución software completa y adaptable a distintos escenarios reales. Desde los fundamentos de la ingeniería de software hasta el manejo de datos en tiempo real y los desafíos a escala urbana, el proyecto demuestra cómo una solución técnica bien diseñada puede responder a las necesidades inmediatas de la comunidad, al tiempo que establece una base flexible para futuras mejoras.

Contents

Chapter 1

Introduction

1.1 Context

Rapidly growing populations are posing significant challenges to urban waste management worldwide. Many cities still rely on outdated waste management practices. These systems struggle to meet growing demand, resulting in financial strain and environmental degradation.

Traditional waste collection systems rely on fixed schedules instead of adapting to demand. This approach often leads to inefficiencies, such as overflowing bins in some locations while others are emptied unnecessarily. These inefficiencies increase fuel consumption, labor hours, and overall costs, while degrading the appearance of public spaces. Furthermore, conventional systems lack access to real-time data on waste generation patterns, making it difficult for municipal authorities to allocate resources effectively and respond to immediate needs.

Environmental concerns add to the urgency for change. Poorly planned collection routes result in excessive carbon emissions from waste collection vehicles, thereby contributing to climate change and undermining the sustainability of current waste management practices.

The development of robust informatics infrastructures to support *Internet of Things* (IoT) technology offers a transformative solution for waste management. By enabling seamless data integration, communication, and processing, these systems allow real-time monitoring of waste bin fill levels and optimization of collection routes. Efficient data flow from sensors to decision-making platforms ensures informed and dynamic operational adjustments.

A well-designed informatics framework enhances efficiency, reduces costs, and supports scalability, enabling cities to improve cleanliness while meeting the demands of population growth and environmental sustainability.

1.2 State of the art

The rapid advancement of IoT technologies has revolutionized urban waste management, addressing long-standing inefficiencies and paving the way for sustainable cities.

Many traditional methods rely on static collection schedules and lack real-time visibility into how full each container is. This lack of information often leads to inefficiencies, such as overflowing bins in high-demand areas or premature pickups of half-empty containers in lower-demand zones. Addressing these issues, researchers and companies have started to develop more dynamic, data-driven systems that integrate continuous monitoring and computational intelligence.

Smart waste management, sometimes referred to as *Intelligent Waste Management* (IWM), leverages IoT sensors, wireless communication, data analytics, and automated decision-making algorithms. Current approaches aim to achieve four main goals:

1. Real-time monitoring of container fill levels and conditions.
2. Facilitating on-demand route planning and optimization
3. Minimal environmental impact through reduced fuel consumption and optimized collection frequency.
4. Predictive analytics to anticipate future waste patterns

Continuous Monitoring and Computational Intelligence

IoT-based systems use sensors such as ultrasonic transducers for real-time measurement of container fill levels, or image detection to identify contamination and blockages. On the computational side, machine learning and AI-based approaches frequently use *k-means* clustering for grouping containers with similar usage patterns, while metaheuristics (e.g., genetic algorithms) and standard *Travelling Salesman Problem* (TSP) heuristics are often applied to optimize vehicle routing under varying constraints.

IoT-based waste monitoring systems rely on three pillars: hardware, telecommunication networks, and informatics infrastructure. Hardware includes smart sensors and microcontrollers in waste containers to track fill levels, temperature, and contamination, generating real-time data on waste conditions. Technologies like 5G enable low-power, long-range data transfer to centralized systems..

The informatics infrastructure manages and analyzes this sensor data using scalable cloud platforms for storage, processing, and analytics. Efficient data pipelines handle frequent streams with minimal delay. Cloud systems enable predictive insights, such as forecasting full containers, identifying patterns, and optimizing routes through machine learning. Real-time dashboards and alerts help address issues like overflowing containers or fire risks.

Advancements in research have propelled real-time processing and communication, with *edge computing* emerging as a key solution. By processing data locally at the source instead of transmitting it to a centralized cloud, edge computing reduces network strain and minimizes latency.

These advancements are driving smarter, more efficient waste management, contributing to urban sustainability and resource optimization.

Academic and Industry Initiatives

Numerous institutions and companies are advancing IoT-based waste management. For instance, the Sentilo Platform [1], initiated by the Barcelona City Council, has demonstrated the potential of large-scale sensor deployment with adaptable software tailored for municipalities.

On the commercial front, solutions like Bigbelly [2] offer smart bins equipped with solar-powered waste compactors and fill-level alerts, while other firms provide comprehensive systems that integrate sensors, cloud-based optimization engines, and hardware. These innovations address urban waste management challenges by improving efficiency and reducing environmental impacts.

In addition, university research has played a vital role in developing innovative waste management solutions. Gast et al.[3] proposed an adaptive algorithm for real-time waste collection routing, optimizing logistics to improve efficiency.

Together, these efforts—spanning municipal initiatives, commercial solutions, and academic research—highlight the growing role of smart technologies in promoting sustainable and effective urban waste management practices.

Current Trends and Challenges

While recent work points to significant successes, a few challenges remain:

- *Standards and Interoperability*: Proprietary systems can limit scalability and integration, researchers and industry leaders are advocating for open standards and protocols that ensure different systems can work together seamlessly [4]. Open standards not only enhance interoperability but also stimulate innovation by allowing broader participation from developers and smaller companies.
- *Scalability and Cost-Effectiveness*: Large cities require systems that handle massive sensor networks without overburdening budgets. Edge computing strategies are being explored to decrease bandwidth and computational overhead, keeping operational costs feasible.
- *Institutional Support and Wider Adoption*: Institutional backing is essential for the long-term success of these initiatives. While IoT-based waste management has demonstrated transformative potential, sustained investment and further development are

required. Policies should aim to bridge the gap between highly developed urban centers and smaller or less developed cities, ensuring equitable access to technological advancements.

Summary

State-of-the-art waste management is increasingly focused on combining IoT sensors, advanced information systems, and AI-driven decision-making. Sensors provide real-time alerts on container fill levels, while AI-powered algorithms optimize routes and adjust in real time. Machine learning adds value by predicting trends and detecting issues, making systems more efficient. These innovations promise cost savings, lower emissions, and better sanitation, but challenges like technology limitations, costs, and regulations still hinder widespread adoption.

Looking ahead, the future of smart waste management will likely center on open and modular platforms, stronger wireless networks, and better integration with urban planning and environmental policies. As technology advances in AI, communication networks, and cloud-edge computing, smart waste management can play a pivotal role in shaping sustainable cities, providing cleaner, more efficient urban environments while mitigating environmental impact.

1.3 Personal motivation

The primary motivation behind this project is to apply the advanced knowledge and skills acquired during the academic journey in areas such as Software Design, Distributed Software, and Software Engineering to deliver a professional product that adheres to industry standards. By focusing on strong architectural practices, collaborative methodologies, and robust development strategies, the project aims to create an efficient, maintainable, and user-friendly system that meets the highest professional standards.

Additionally, the project seeks to address the challenges of acoustic and visual pollution caused by traditional waste collection systems. Overflowing trash containers not only inconvenience residents but also detract from the aesthetic appeal of urban areas, particularly those with historic and cultural significance. Noise and disruptions caused by garbage trucks, especially during nighttime collections, can negatively impact residents' quality of life. By implementing smarter and more efficient waste management methods, this project aims to foster quieter and cleaner urban environments.

Finally, this initiative embraces a long-term vision of continuous innovation. IoT-based waste management systems hold significant potential for iterative improvement by integrating advancements in data analysis, algorithms, and hardware. This approach ensures adaptability and relevance over time, laying the groundwork for future developments. Through this foundation, the project seeks to contribute to the ongoing evolution of smarter, more sustainable urban waste management systems.

Chapter 2

Proposed Solution - Higiea

Building on the issues highlighted in previous chapters—ranging from inefficient collection routes to limited *real-time visibility*—this chapter presents *Higiea*, a *modular waste management platform* designed for contemporary urban demands.

We begin by reviewing the system's core features, emphasizing how it interacts with IoT sensors, processing pipelines, and route management. We then discuss how Higiea addresses scalability, integration, and future-readiness, before concluding with the high-level objectives that guide the platform's design.

2.1 High-Level Concept

Higiea aims to modernize municipal waste collection by offering a robust, *end-to-end informatics infrastructure* that integrates *real-time waste container monitoring*, automated route management, and historical data for previous analysis. Higiea eliminates reliance on fixed collection schedules by leveraging *IoT sensors* to capture container fill levels, transmitting this data to a centralized platform in near real time.

Higiea employs scalable cloud infrastructure to ingest, process, and store sensor data, ensuring reliability and performance even in large urban deployments. Communication is based on standard protocols, preserving flexibility and modularity. By focusing on message protocol compatibility, Higiea ensures that it can efficiently receive and process messages from diverse sensors technologies, such as weight sensors, proximity detection sensors, or others, adapting to the unique needs of different municipalities.

Table 2.1: Core Features and Benefits of Higiea

Feature	**Description**	**Benefit**
Real-time Monitoring	IoT sensors provide *container fill-level data* in near real time.	Enables real-time monitoring and visualization of trash container statuses, facilitating efficient management and timely responses.
Automated Route Management	Rules-based engine triggers new routes when thresholds are reached.	Reduces manual oversight, prevents overflows, and minimizes unnecessary pickups.
Historical Analytics	Logs *sensor updates*, truck routes, and system decisions.	Enables long-term planning and *trend analysis* to optimize resource allocation.
Scalability	Cloud-based or on-premise solution that handle growing data and user needs.	Ensures suitability for both small towns and large metropolitan areas.

2.2 Main Objectives

- **Deliver a Production-Ready Platform:** Develop an application robust enough for *real-world municipal waste collection*, emphasizing reliability, security, and ease of maintenance.
- **Implement Realistic IoT Protocols:** Integrate real-life IoT protocols to enable the reception of sensor data from actual field devices, ensuring seamless future deployment.
- **Automate Route Management Processes:** Facilitate automatic detection of *waste container fill levels* and dynamic generation of collection routes with minimal manual intervention.
- **Maintain Comprehensive Container and Route Tracking:** Ensure an accurate, *real-time record* of each container's status and every route assigned to the *collection fleet*.

2.3 Secondary Objectives

- **Scalability and Versatility:** Design the solution to scale seamlessly from small towns to large urban areas, accommodating growing workloads and evolving user needs.
- **Modularity and Ease of Adaptation:** Design the system so that major components (e.g., trigger mechanism, route planning, route calculation) can be replaced or upgraded independently.

- **Technology-Decoupled Design:** Avoid reliance on specific or proprietary technologies, enabling integration with a wide range of sensor devices and communication protocols.
- **Minimal Setup:** Facilitate rapid deployment and configuration, reducing the time and expertise required to bring the system online.
- **Future-Readiness:** Establish a foundation capable of integrating advanced features, including *AI-driven route optimization*, traffic-aware routing, and *predictive maintenance*.
- **Maintain Clean Code and Architecture:** Adhere to coding and design best practices, ensuring that new developers or stakeholders can quickly adapt and extend the system.

2.4 Expected Outcomes

Achieving immediate *production viability* is a primary outcome, enabled by the implementation of realistic IoT protocols and a decoupled architecture. By ensuring that, the final application should be deployable without delay, allowing municipal waste managers to begin realizing efficiencies and cost-savings right away, with minimal adjustments needed to existing workflows. The *production-readiness* should also enable quick scaling from small pilot programs to city-wide environments, allowing municipalities to incrementally expand their waste collection capabilities.

Another important outcome is the enhanced monitoring of container statuses and the collection routes assigned to service vehicles. The system's *real-time visibility* into sensor data for container fill levels and its documentation of routes aims to offer operational transparency for stakeholders. Real-time tracking of truck routes and container capacities is expected to facilitate data-driven decision-making, prevent overflows, and reduce unnecessary pickups.

A further outcome centers on the *automation* and *adaptability* of the waste collection workflow. Through automated triggers and route planning, collection schedules are going to be adjustable dynamically based on actual container fill levels, thereby reducing manual oversight. This streamlined process should not only boost operational efficiency but also empower municipal administrations or service providers to tailor their collection strategies to local requirements. By merging these abilities with a *modular design*, the solution must ensure flexibility, making it simple for different jurisdictions to adopt or refine route calculation approaches according to varying urban layouts and constraints.

Finally, the platform is designed to establish a robust foundation for advanced feature integration. As the baseline system already includes real-time monitoring, container status historical data, and automated route management, future innovations—such as p*redictive maintenance* for containers or *AI-driven* route optimization—should be readily

incorporable. This future-ready structure ensures cities remain adaptable and prepared to address evolving waste management challenges, including population growth, changing environmental regulations, and advancements in IoT technologies.

Chapter 3

Planning

This chapter presents the initial planning of the project and how it was adapted throughout its development. A *Gantt chart* (commonly used to visualize timelines and dependencies) was created to illustrate the original schedule alongside the final revised schedule. The final plan reflects real-world adjustments made in response to technical challenges, scope decisions, and the need for continuous integration and deployment processes.

The planning was approached to ensure each critical component—infrastructure, backend, route optimization, IoT integration, frontend and testing—received focused attention while outlining the main critical points.

3.1 Initial Planning and Phases

The original plan consisted of *seven main phases*, each with a clear objective and an estimated duration. Table 3.1 summarizes the phases, the estimated days, and a brief description of the key points of each phase.

Table 3.1: Initial Project Planning

Phase	Duration (Days)	Key Tasks
Phase 1: *Tech Study & Planning*	15	• Research various technologies and methodologies. • Define overarching *system architecture* and core modules. • Clarify *functional and non-functional requirements*.
Phase 2: *DevOps Setup*	15	• Build a minimal prototype to validate *deployment workflows*. • Configure an environment for *continuous integration* and *deployment* (*CI/CD*).
Phase 3: *Backend Development*	20	• Implement main domain logic and necessary endpoints. • Integrate with a suitable database for data persistence. • Establish *logging*, error-handling, and security protocols.
Phase 4: *Route Optimization*	15	• Develop or integrate a route optimizer (e.g., based on the *Traveling Salesman Problem*). • Ensure support for capacity constraints and sensor data handling. • Test and refine algorithms for performance and accuracy.
Phase 5: *IoT Implementation*	15	• Simulate sensor data ingestion and processing. • Implement *message-driven* components for real-time updates.
Phase 6: *Frontend*	15	• Develop a graphical user interface for system interaction. • Integrate *UI* features with the backend services in real time.
Phase 7: *Testing & Delivery*	15	• Execute final tests (performance, security, reliability). • Address any identified issues and refine documentation. • Prepare deliverables for presentation and final review.

3.2 Final Planning and Deviations

While the initial plan provided a solid framework, several adjustments were necessary due to *real-world constraints* and technical decisions:

- **Fase 1 and Fase 2** were completed largely as planned, providing a clear technology stack and foundational DevOps environment.
- **Fase 3 and 4 Overlap:** We mixed some DevOps tasks with backend and route optimization tasks, especially since we relied on a third-party service to compute optimal routes. This integration required additional infrastructure work.
- **Fase 5 Success:** IoT integration and security protocols were completed smoothly, aligning with the estimated schedule.
- **Fase 6 and 7 Reduction:** The time allocated to the final frontend and testing was partially reduced to accommodate more effort on the thesis and documentation. Consequently, the frontend remained simpler (showing only essential functionality).

Additional *challenges* arose from:

- Significant infrastructure deployment, database setup, and server configurations, causing certain tasks to require more time than anticipated.
- The decision to use a *third-party* route calculator instead of fully developing our own *TSP solution* from scratch. Developing a robust custom route optimizer could be a separate, large-scale project in its own right.
- Testing complexities in a distributed services environment. Comprehensive *End-to-End* (E2E) testing took longer due to multiple components needing seamless integration.
- Documentation and Thesis Work: The final days often entail concurrency between delivering features (phase 7) and polishing or documenting the system for academic submission.

Meetings with Supervisor

After each major phase or iteration (e.g., technology study, infrastructure deployment, backend completion, route algorithm integration), a dedicated meeting with the tutor was held to:

- Present the outcomes and discuss any newly discovered *risks* or *constraints*.
- Validate progress against the project objectives.
- Refine upcoming tasks or pivot the approach, if necessary.

This iterative process helped keep the project aligned with academic and technical goals.

3.3 Gantt Chart

Figure 3.1 presents the Gantt diagram outlining the six main phases of the project:

Planification, *DevOps Setup*, *Backend Development*, *Route Calculation*, *IoT Integration*, *Simulator Development*, and *Evaluation*.

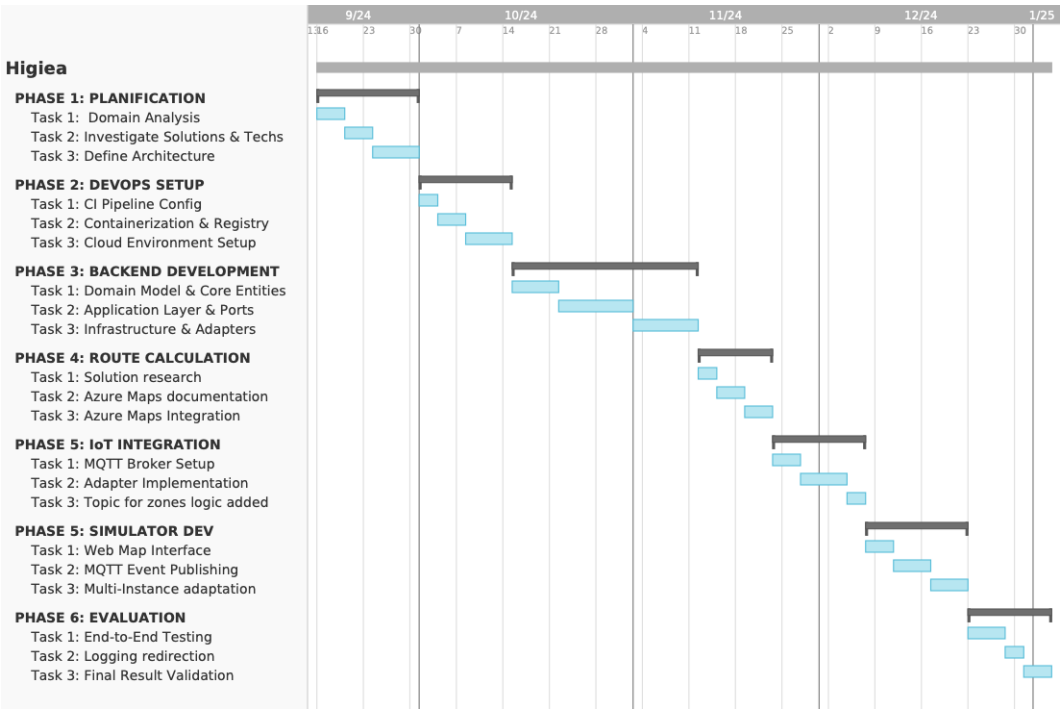

Figure 3.1: Gantt Chart for the Project Plan

Each phase comprises multiple tasks (e.g., domain analysis, CI pipeline configuration, Azure Maps integration, etc.), illustrated by individual bars that indicate start and end dates. The darker bars represent *higher-level* phases, while the lighter bars depict their *sub-tasks*.

This structure provides a clear visual progression of the project timeline, highlighting dependencies among phases and serves as a high-level roadmap, ensuring that all critical components are addressed within the designated time frames.

3.4 Methodology

The project was hosted on *GitHub*[5], where a *Kanban* board was used to manage tasks and maintain visibility on progress and backlog items. This approach was inspired by

Agile principles [6], though the decision was made to forego a *Scrum*-like [7] sprint framework. While sprint planning, iterations, and retrospectives are highly effective in team environments, they were deemed to be more time-intensive than beneficial for the scope of this individual project. The flexibility of Kanban allowed for continuous prioritization and task management without the overhead of formalized sprints.

A branching strategy was implemented to ensure efficient code management, with two primary branches: `main` for production and `dev` for integration and testing. Changes were merged through *Pull Requests* (PRs), which required code reviews and passing automated tests before integration. *GitHub Actions* powered the *Continuous Integration*(CI) workflows, automatically triggering tests for each PR. This ensured that no changes could be merged into the `dev` or `main` branches if tests failed.

For *Continuous Deployment*(CD), an automated pipeline deployed the `main` branch to the production environment whenever a new *release* was created. To maintain consistency and clarity in tracking changes, the project followed Semantic Versioning (`vX.Y.Z`)[8]. In this scheme, the major version (`X`) was incremented for backward-incompatible changes, the minor version (`Y`) for backward-compatible new features, and the patch version (`Z`) for bug fixes and minor improvements.

A visual representation of the branching and release strategy followed during the project development is shown in Figure 3.2.

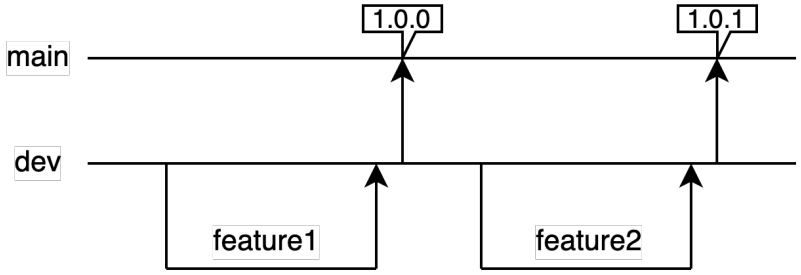

Figure 3.2: Branch and Release Strategy followed during the development of Higiea

Reflections on the Planning Process

Overall, the planning served as a critical guide but required flexibility to accommodate new information, infrastructural dependencies, and third-party services. The final outcome demonstrates how a structured plan, combined with iterative refinements, can adapt to real-world challenges while still achieving a successful result. The project's *DevOps* environment, simplified frontend, and reliance on a proven third-party route calculator ensured a stable and functional platform within the defined scope and timeline.

Chapter 4

System Analysis

This chapter explores the analytical background of Higiea including its domain model, central ideas, and requirements. By examining the interactions between sensors, trucks, and routes, we define the grounds of the system and how it guarantees waste collection and operational efficiency.

We begin with an overview of the domain model, describing the roles and relationships of key entities. It then explores the mechanisms for triggering, planning, and calculating routes, emphasizing the modularity that allows for easy adaptation to different operational policies or scenarios.

A detailed presentation of the system's both functional and non-functional requirements is then provided, concluding with use cases that demonstrate how the system responds to various real-world situations.

4.1 Domain Model and Concepts

This section provides a detailed look at the core entities and processes in Higiea. The focus is on how sensors, trucks, and routes interact to ensure waste containers are not full for extended periods of time. The design philosophy revolves around timely updates from sensors, automated route triggers, and flexible planning strategies to accommodate different city sizes and operational constraints.

Overview of the Domain

Three primary components form the backbone of the system:

- **Sensors**:
  - Each sensor is mounted on a **waste container**. It monitors the container's fill level, which can be *EMPTY*, *HALF-FULL*, or *FULL*.

  - A sensor is uniquely identified by an *ID* and has a recorded *GPS coordinates* location for accurate mapping and route calculations.
  - When the container's fill level changes, the sensor transmits a message to the system, which then updates the sensor's state in real time.

- **Trucks**:
  - Each truck is based at a specific depot, which serves as both the start and end point for its routes.
  - A key attribute of a truck is its *maximum load capacity*. This capacity influences how many waste containers it can collect during a single route.
  - Once a route is completed, the truck notifies the system of its availability. This allows for flexible assignment and rapid redeployment if there are additional pending collections.

- **Routes**:
  - A route is automatically triggered based on a route trigger decision defined by the system rules whenever specific conditions are met.
  - A route's *planning stage* determines which sensors should be collected and which truck will serve them.
  - The calculation stage then generates the path for the assigned truck, starting and ending at its depot.

Figure 4.1 illustrates how the principal entities —Sensor, Truck, Route, Location— relate to each other in the Higiea system's domain model.

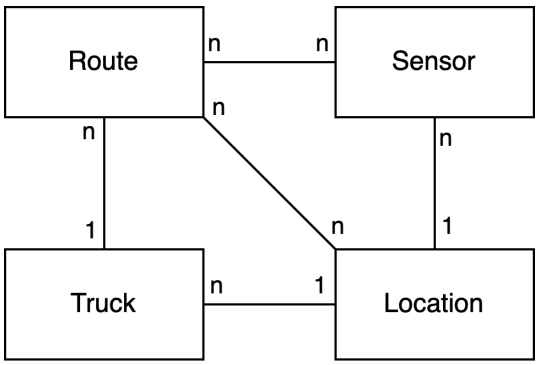

Figure 4.1: Domain Model

When a sensor updates its state, the system reacts by evaluating whether a route trigger is necessary. If triggered, the planning mechanism selects a suitable truck (or trucks), determines the subset of sensors to be collected, and instructs the route calculator to find the optimal path.

4.2 Route Trigger, Planning, and Calculation

Although the system supports modular and customizable strategies, we follow a specific approach to triggering, planning, and calculating routes. This approach can be adapted in the future to suit different operational scenarios or policy requirements.

Route Trigger

A route is triggered under these conditions:

- A sensor transitions to the *FULL* state.
- There is at least one available truck (i.e., not currently assigned to another route).

This ensures that *full containers* are addressed promptly, reducing the likelihood of overflows and maintaining cleanliness.

Route Planning

The system's planning mechanism prioritizes:

- **Full Sensors Over Half-Full Sensors:** *FULL* containers are critical and must always be included. *HALF-FULL* waste containers are collected only if there is enough capacity, ensuring the truck is not overloaded
- **Minimum Sufficient Truck Capacity:** Among the available trucks, the system attempts to find the one with the *smallest capacity* that can handle all of the targets. This avoids unnecessarily assigning a large-capacity truck to a small set of containers.
- **Fallback to Largest Truck:** If no single truck can accommodate all *FULL* and *HALF-FULL* containers, the system picks the truck with the largest capacity. It then removes *HALF-FULL* sensors until the truck's capacity is no longer exceeded.

By systematically selecting smaller trucks when possible and deprioritizing *HALF-FULL* trash containers if necessary, this strategy balances efficiency and responsiveness.

Route Calculation

Once the sensors are selected and a truck is assigned, the system determines the shortest possible route. This route starts at the truck's depot, visits each sensor in an optimal sequence, and returns to the depot, effectively solving a *Traveling Salesperson Problem*.

By following this approach, the system *minimizes fuel consumption, travel time, and operational costs* while ensuring that all assigned sensors are serviced efficiently.

4.3 Requirements Specification

This section details the system's requirements. *Functional Requirements* (FR) specify the features and behaviors the system must provide, while *Non-Functional Requirements* (NFR) describe the *performance*, *reliability*, and *scalability* constraints.

4.3.1 Functional Requirements

FR1 Sensor State Updates: Sensors must be capable of sending updates to the application regarding the state of their trash containers. The application must then update the corresponding sensor's state in its internal database.

FR2 Route Trigger Decision: A new route must be triggered whenever a sensor transitions to *FULL*. This requirement ensures a timely reaction, avoiding extended periods of uncollected waste.

FR3 Route Planning Decision: After a route trigger, the system must plan a route assigning a set of sensors based on priority and attempt to fit these sensors into the truck with the smallest capacity sufficient to handle the total load.

FR4 Route Calculation: For any given set of sensors, the system must calculate the *shortest route* starting from the truck's depot, visiting each sensor, and returning to the depot.

FR5 Logs of State Changes and Routes: The system must maintain comprehensive *logs* for each *sensor's state change*, including the time, sensor ID, and new state, as well as for each route creation event, recording the involved truck, assigned sensors, and timestamp.

FR6 Route Updates: Once a truck finishes its assigned route, it must be able to *notify* the system so that it can be dispatched to a new route if required.

4.3.2 Non-Functional Requirements

NFR1 Performance: Sensor updates must be processed quickly—ideally within a few seconds—so that route creation is not delayed. This is especially crucial in dense urban areas where many containers might fill up simultaneously.

NFR2 Modularity: The triggering, planning, and calculation components must remain modular. This design supports easy replacement (e.g., switching from a basic route calculator to a *IA-based* one) without changing the entire system.

NFR3 Scalability: The system must efficiently manage a large number of sensors, reliably processing frequent *state-change messages* under peak load conditions. Horizontal scaling should be possible to handle expanding infrastructure.

NFR4 Reliability: The logging and state-tracking mechanisms must be accurate and robust. Inconsistent logs or missed sensor updates could lead to overflows or inefficient collections.

NFR5 Adaptability: The same application should serve both small municipalities and large metropolitan centers. This adaptability is a core benefit, ensuring the system can grow alongside a city's changing needs.

NFR6 Maintainability: The codebase should be clean and logically structured. Clear separation of concerns ensures developers can adapt or extend the system without excessive refactoring.

4.4 Use Cases and Scenarios

Below are the primary use cases illustrating how the application reacts to sensor updates and manages route creation, planning, and calculation. The presented scenarios capture day-to-day operational flows as well as exceptional events (e.g., sudden overflows).

Use Case 1: Sensor Status Monitoring

Goal: Allow system administrators to view current sensor statuses in real time.
Actors: System Administrator (viewer), Application (provider).
Description:

1. The administrator enters the *monitoring interface*.
2. The system retrieves the latest sensor states (i.e., *empty*, *half-full*, *full*) from the database.
3. The dashboard displays each sensor's ID, fill level, and location on a map.

Use Case 2: Sensor State Update

Goal: Keep sensor states current whenever fill levels change.
Actors: Sensor (initiator), Application (receiver).
Description:

1. The sensor detects a change (e.g., *empty* to *half-full*, *half-full* to *full*).
2. The sensor communicates the change, including its unique ID and new *fill state*, to the application.
3. The application processes the information, updating the sensor state in its database and recording the event in an event log.
4. If the sensor state indicates that it is now *full*, the system triggers the route planning process (see Use Case 3).

Use Case 3: Route Trigger on Full Sensor

Goal: Immediately generate a new route when a sensor becomes *full*.
Actors: Sensor (initiates), Application (receives).
Description:

1. A sensor transitions from *half-full* to *full* and sends an update.
2. The application logs the event and checks whether a route trigger is required.
3. The system initiates a new r*oute creation sequence*, invoking the *planning module*.

Use Case 4: Route Planning

Goal: Decide which sensors to collect and which truck to use based on capacity and current sensor states.
Actors: Application.
Description:

1. The application compiles a list of all sensors that are *full* or *half-full*.
2. The *planning algorithm* attempts to fit these sensors into the smallest-capacity truck capable of carrying the total load.
3. If no single truck can accommodate them all, the system picks the truck with the largest capacity, removing some *half-full* sensors to ensure load feasibility.
4. The system assigns the resulting list of sensors to the selected truck, logs this assignment, and calls the route calculation module.

Use Case 5: Route Calculation

Goal: Compute the shortest route for the assigned truck and sensors.
Actors: Application.
Description:

1. The `RouteCalculator` takes the truck's depot location and the list of sensors as input.
2. It applies a suitable routing algorithm (e.g., shortest path, *traveling salesman heuristics*) to determine the best sequence of sensor visits.
3. The final route (including travel order and distance) is stored and associated with the truck. A route creation event is logged.
4. The truck operator can then follow these directions in real-time, or via dispatch instructions, to collect each sensor in the specified order.

Summary

The System presented combines a thorough view of the domain model with well-defined functional and non-functional requirements. By focusing on modular triggers, planning strategies, and route calculation mechanisms, the design remains flexible and scalable.

The included use cases demonstrate how real-time sensor state changes lead to automated route generation, ensuring containers are serviced before they overflow and that operational resources (trucks and drivers) are utilized effectively. This modular, data-driven approach supports municipalities of varying sizes and is readily extensible to evolving waste collection technologies or policy changes.

The next chapters will discuss how this system has been implemented, starting with an exploration of the theoretical background to provide foundational insights into the technologies and principles essential for understanding it.

Chapter 5

Theoretical Background

This chapter provides all relevant theoretical concepts and foundational knowledge required to understand and implement our waste management system, Higiea. Readers should first become familiar with these ideas before proceeding to Chapter 6 (System Architecture) and Chapter 7 (Implementation).

5.1 Hexagonal (Ports & Adapters) Architecture

Our project's design follows *Hexagonal Architecture* design pattern [9], also known as the *Ports and Adapters* pattern. Introduced by Alistair Cockburn, this pattern emphasizes the creation of applications that are *flexible*, *maintainable*, and *independent* of specific *external technologies*.

Hexagonal Architecture focuses on separating the *core* of the application—the domain logic and business rules that define the application's essential functionality—from external dependencies. In other words, it aims to completely isolate the business logic, making it independent of any external implementation details such as database engines, user interfaces, or third-party services.

To achieve this, it defines a layered structure with well-defined responsibilities for each layer, designed to be self-contained and without knowledge of the details or implementations of the external layers interacting with them.

Hexagonal Architecture separates the application into *domain*, *application*, and *infrastructure* layers. Figure 5.1 illustrates the high-level layout of the pattern:

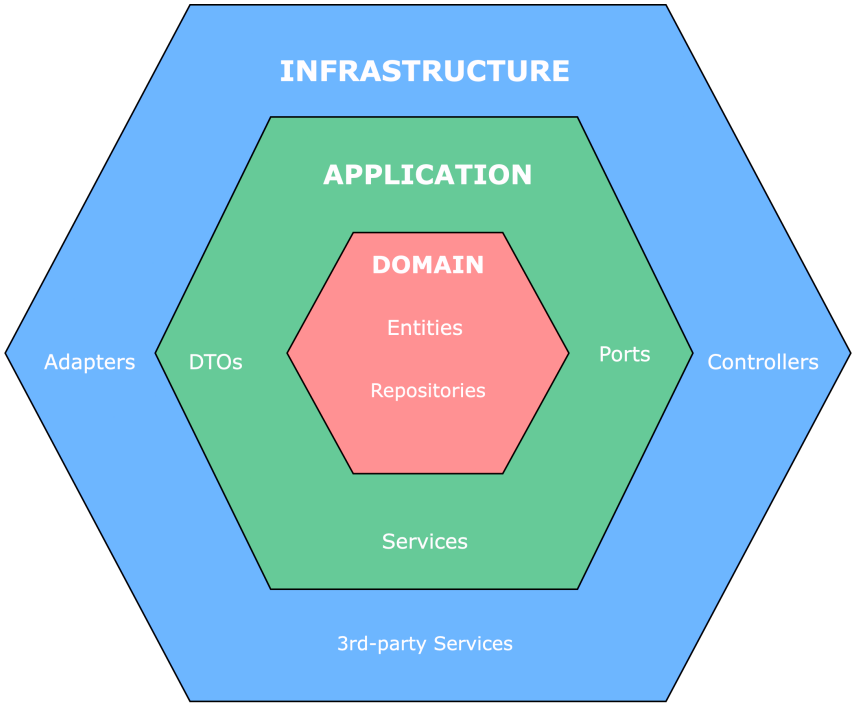

Figure 5.1: High-Level View of Hexagonal Architecture

- **Domain Core:** Encapsulates business rules, including entity models and constraints.
- **Application Layer:** Defines use cases and orchestrates business logic through *ports*, which serve as abstractions for external interactions. The goal of this layer is to process requests from the infrastructure, invoke domain logic, and return responses while maintaining decoupled communication.
- **Infrastructure Layer:** Handles interactions with external systems, such as databases, APIs, or messaging systems. Through *adapters*, it provides concrete implementations of the ports defined in the application layer, enabling integration with external dependencies.

Ports, as defined in the application layer, serve as *interfaces* for communication, operating on *abstractions* rather than specific implementations. Adapters in the infrastructure layer implement these ports, enabling interaction with external systems while keeping the application logic isolated from infrastructure details.

Key Features of the Hexagonal Pattern

- **Core Independence:** The domain logic remains free from external technology concerns.
- **Bidirectional Communication:** Well-defined ports handle both inbound (user input, sensor data) and outbound (database queries, external APIs) requests.

- **Testability:** Because external services are decoupled through ports, the domain can be tested using mocks or stubs.

- Hexagonal Architecture **adheres to the SOLID Principles [10]** , particularly:

  – *Single Responsibility Principle* (SRP): By separating domain logic, application orchestration, and infrastructure concerns, each layer has a clear and distinct responsibility.

  – *Dependency Inversion Principle* (DIP): Maintains dependence on abstractions, rather than concrete implementations.

- **Separation of Concerns & Modularity:** Clearly delineated domain, application, and infrastructure layers promote easier maintenance and reuse.

Flow and Interaction

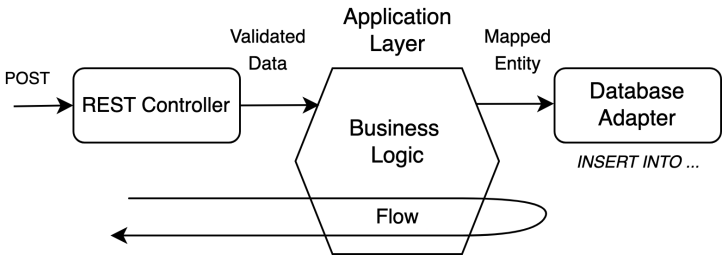

Figure 5.2: Flow of Hexagonal Architecture

As illustrated in Figure 5.2, the flow for handling a `POST` request begins when the request first reaches the *controller* in the infrastructure layer, it validates the input data and delegates the operation to the appropriate application service within the application layer.

This layer orchestrates the necessary steps to complete the operation. For instance, in the case of creating a new entity, this orchestration might include verifying that the entity does not already exist, ensuring compliance with business rules or constraints, creating the entity, and saving it to the repository.

When saving the entity, the application service interacts with the *repository* port [11], an abstract interface that defines the operations required to persist data. The *repository port* is implemented by an adapter in the infrastructure layer, tailored to a *specific database* or *storage system*. This adapter handles the actual persistence logic, translating the core's requests into database-specific commands.

5.2 Messaging and MQTT

Message Queuing Telemetry Transport (MQTT) [12] is a lightweight messaging protocol designed for constrained devices and low-bandwidth, high-latency, or unreliable networks. It follows a *pub-sub* pattern, meaning:

- *Publishers*: Entities (e.g., sensors) that publish messages to specific topics.
- *Subscribers*: Entities (e.g., our application) that subscribe to these topics to receive messages.
- *Broker*: A central server that relays messages from publishers to subscribers, ensuring decoupled communication.

This structure contrasts with *HTTP*'s [13] *request-response* approach, where sensors would need to call an endpoint repeatedly, often leading to increased overhead, potential congestion, and complexity in load distribution.

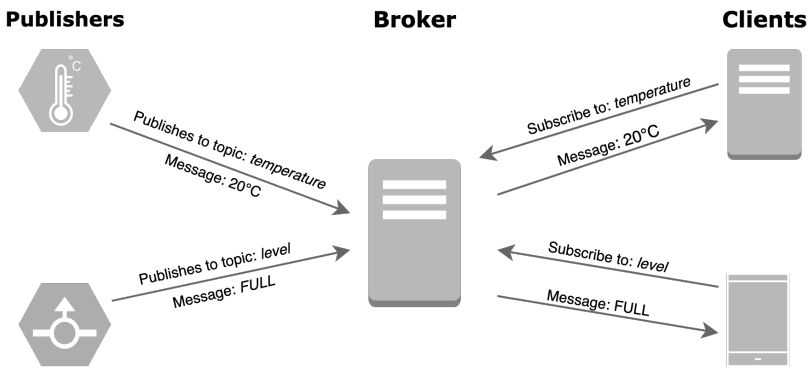

Figure 5.3: MQTT Publish / Subscribe Architecture

Figure 5.3 shows how the MQTT Publish/Subscribe Architecture, where:

- Sensors simply publish state updates (e.g., container fill level) to a broker topic.
- The broker is responsible for routing these messages to all active subscribers for that particular topic.
- Our application only processes incoming messages when they arrive, avoiding unnecessary polling.

MQTT is a standard in the IoT industry due to its lightweight design, making it ideal for *resource-constrained* devices and networks. It enables efficient communication in scenarios with large numbers of devices sending frequent but small messages, low or variable network connectivity ensuring robust message delivery despite unreliable conditions, and the need for event-driven architecture that allows systems to respond to sensor updates immediately.

Additionally, its support for *Quality of Service* (QoS) levels, retained messages, and last will/testament features further strengthens its reliability and suitability for IoT applications.

For these reasons and after evaluating protocol options, we concluded MQTT offers the best trade-off for reliability and resource constraints.

5.3 Reactive Programming

Reactive programming[14] is a software paradigm that focuses on handling data as streams of events, allowing applications to react to changes as they happen instead of following a fixed sequence of steps. This approach is particularly effective for building systems that handle *simultaneous tasks*, such as *real-time data feeds* or *user-heavy web applications*.

The principles of reactive programming are guided by the *Reactive Streams* [15] specification, which defines key interfaces—`Publisher`, `Subscriber`, `Subscription`, and `Processor`—to manage the flow of data between producers and consumers efficiently.

At its core, reactive programming uses two key ideas:

- *Asynchronous* and*Non-Blocking Operations*: Tasks don't wait for each other to finish. For example, if you're waiting for data from a server, your program can keep working on other tasks instead of pausing.

- *Data Streams*: Information flows continuously, and you can work with it as it arrives, piece by piece.

A common concept in Java reactive programming[16] is the `Mono`, representing a stream with either one result or none, often used for tasks like fetching a user by ID. In contrast, a `Stream` handles multiple pieces of data, such as database rows or a flow of events. Both come with tools, called *operators*, to process data as it flows—allowing transformations (e.g., converting numbers to strings), filtering (e.g., skipping unwanted values), and combining multiple streams into one.

By using these ideas, reactive programming helps create systems that are responsive, efficient, and ready to handle modern workloads.

Illustrating a Reactive Data Flow

Figure 5.4 illustrates a high-level overview of how data flows through a reactive pipeline. The diagram emphasizes how messages move from a Service that acts as a *Publisher* (e.g., MQTT broker) into a Subscribed service.

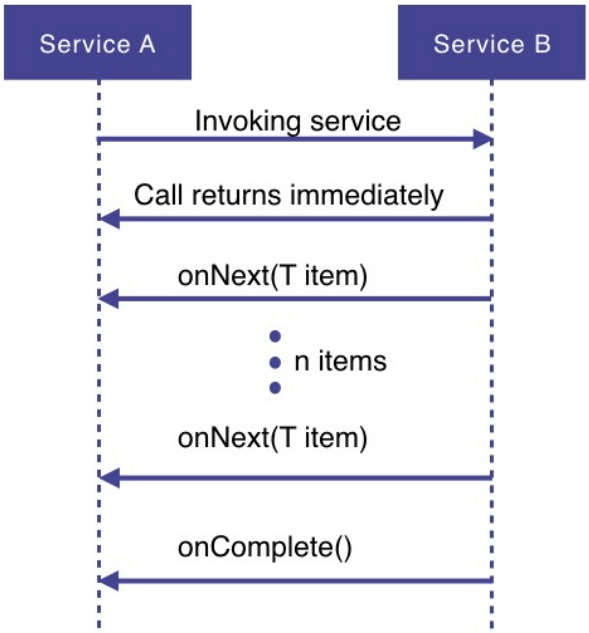

Figure 5.4: High-Level Reactive Pipeline flow.

By using reactive principles, *Service A* remains non-blocking throughout this interaction. It delegates the publishing operation to the *Service B* and immediately frees up its resources for other tasks. This design is crucial in systems requiring high concurrency and real-time data processing, as it ensures scalability and responsiveness even under heavy workloads.

For a high-throughput, event-driven system like *Higiea*—where thousands of sensors may publish updates every minute—the lightweight, non-blocking nature of reactive programming yields significant performance benefits, simplifies scaling, and ensures consistent responsiveness.

Chapter 6

System Architecture

Designing the architecture for a complex and evolving system—especially one integrating IoT devices, multiple databases, and third-party services—requires a focus on flexibility from the outset. In our context, flexibility means the ability to add new features, replace existing modules (such as routing providers or database engines), and adapt to future requirements without compromising the system's overall stability and maintainability.

In this chapter, we present the *architectural design* of our system and its components, showing how we have engineered it to ensure modularity, independence from external frameworks, and easy substitution of critical elements. We begin by providing a high-level view of the overall system architecture. We then delve into the detailed *Backend Architecture*, followed by our approach to *Persistence and Data Handling*, the integration of the MQTT protocol, and conclude with a discussion on deployment strategies, cloud architecture, and scalability.

6.1 Overall System Architecture

This section provides a high-level view of how the main components in Higiea interact, as depicted in Figure 6.1. The diagram highlights the flow of data from sensor devices to the MQTT broker, then into the backend (and its databases), and finally to the simulator that can drive or monitor the system's functionality.

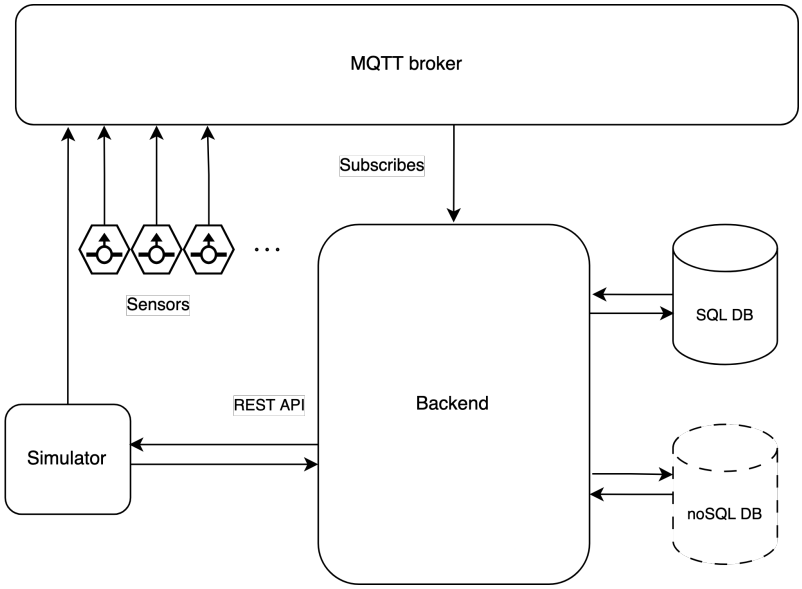

Figure 6.1: High-Level Architecture of Higiea

Key Components

At a high level, the system comprises the following key components:

- *Sensors*: IoT devices attached to trash containers measuring fill levels and location.
- *MQTT Broker*: Central messaging hub facilitating publish-subscribe communication between sensors and the backend.
- *Backend Application*: Core processing unit handling sensor data, updating databases, and managing route planning.
- *Databases*: Dual-database setup using SQL for relational data and NoSql for geospatial data.
- *Simulator*: Optional component for generating synthetic data and testing system functionalities.

In the next sections, each of these components will be discussed in detail to provide a comprehensive understanding of their roles, functionalities, and interactions within the system.

6.2 Backend Architecture

The backend architecture is designed following the Hexagonal (Ports & Adapters) Architecture pattern. This design ensures that our application is flexible, maintainable, and independent of specific external technologies.

Layer Overview

Our system adopts these hexagonal principles by splitting functionality into domain, application, and infrastructure layers (Figure 6.2). This section provides a high-level overview of these layers, while Chapter 7 focus on the specific implementation details of the most important modules.

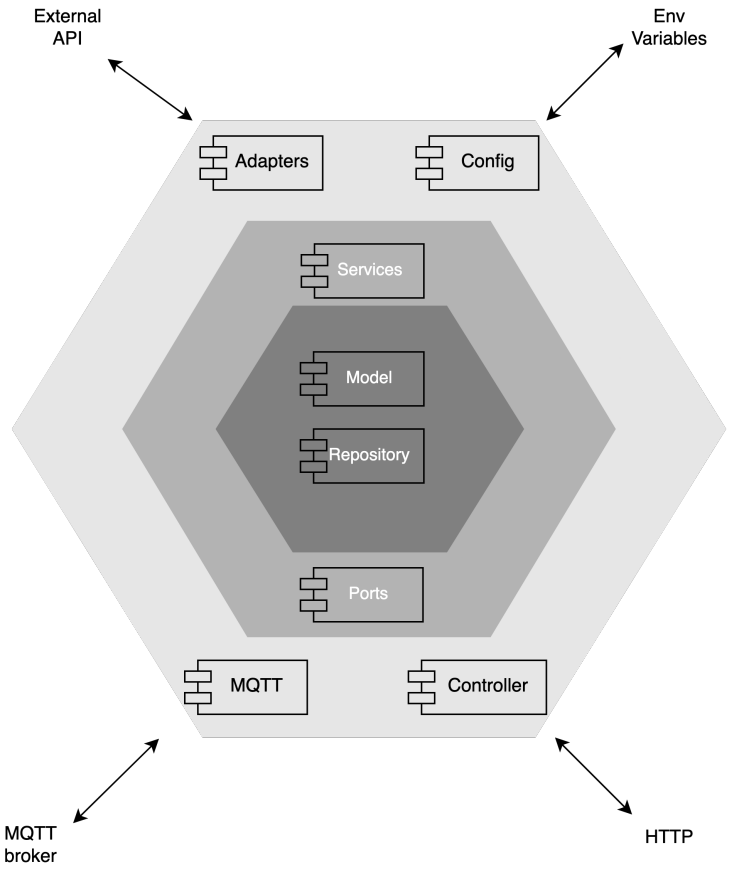

Figure 6.2: Overall Backend Architecture (Hexagonal Pattern)

Domain Layer

At the heart of our system lies the *Domain Layer*, which encapsulates the essential business logic and models. This layer is kept deliberately free of technological or framework-specific dependencies, ensuring that changes to external systems do not jeopardize the integrity of our core business rules. By adhering to this principle, the architecture not only safeguards the domain from external disruptions but also enhances maintainability, testability, and scalability.

The ability to mock repositories and isolate the domain during testing ensures reliable

validation of business rules, while the centralized logic streamlines the integration of new features or external systems, further reinforcing the robustness of the Domain Layer.

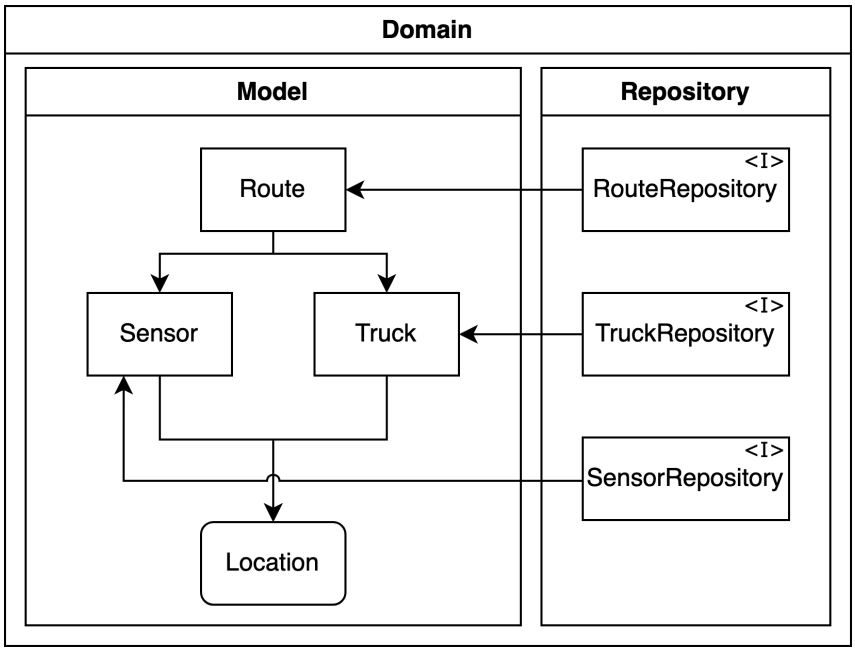

Figure 6.3: Overview of the Domain Layer and Its Core Components

Figure 6.3 shows the two modules of the layer which include:

- **Entities**:
  - *Sensor*: Represents a sensor and encapsulates the business rules related to the entity
  - *Location*: Represents geographical information (points or regions), integral to localization and tracking.
  - *ContainerState*: Maintains key state tied to a sensor asset.
  - *Truck*: Encapsulates transport vehicle attributes, including capacity, availability, and operational constraints.
  - *Route*: Defines the planned or actual paths for a route, facilitating scheduling, optimization, and progress tracking.
- **Repository Interfaces**:
  - Serve as abstraction points for persisting and retrieving entities (e.g., `save`, `delete`, ...).

Application Layer

The *Application Layer* serves as an intermediary between the Domain Layer and external systems, orchestrating operations and workflows. It ensures that business processes are executed correctly and adheres to the principles of modularity and separation of concerns. It contains 6 modules as shown is Figure 6.4:

- **Domain Services:** These services encapsulate reusable business logic operations and serve as the backbone for application-specific workflows. By focusing exclusively on core domain rules, domain services maintain clarity and reusability in the architecture.

- **Services:** These services implement use-case-specific logic, such as processing a sensor message or initiating a new route. They interact with domain services by delegating tasks tied to domain-specific rules, thereby ensuring a clean separation of concerns.

- **Ports:** Ports define abstract interfaces for the services to rely such as different strategies or third-party APIs. We have declared three:

  - *RouteTrigger*: Provides the decision-making logic to determine whether a new route should be triggered.

  - *RoutePlanning*: Defines the overall route plan, including the truck's path and the locations of the garbage containers it must collect.

  - *RouteCalculator*: Determines the optimal order of the container locations within a route.

- **Data Transfer Objects (DTOs):** used to transfer data between different layers or services without exposing internal domain models to the external layers. They ensure data consistency and simplify communication within the system.

- **Requests & Exceptions:** Encapsulate input data for workflows and handle errors cleanly.

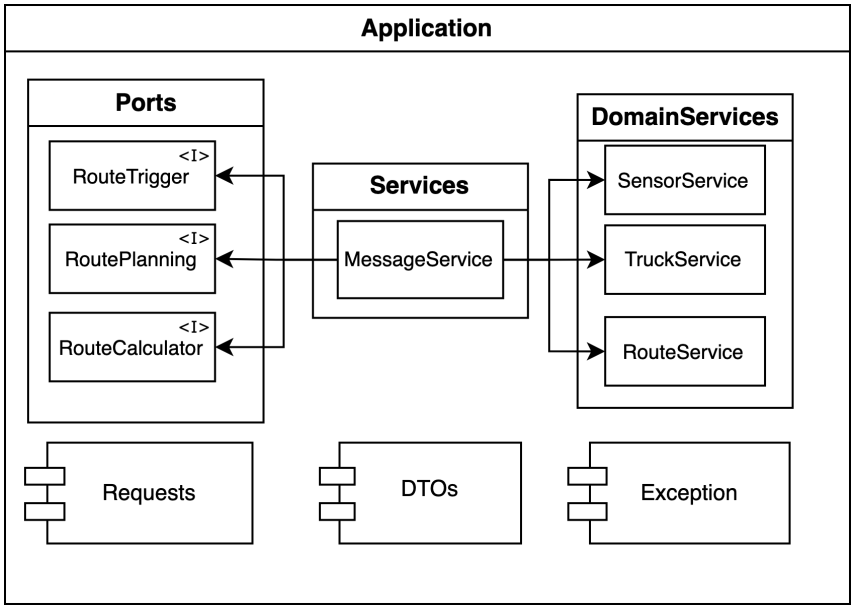

Figure 6.4: Overview of the Application Layer

Infrastructure Layer

The *Infrastructure Layer* handles all interactions with external systems and frameworks, providing concrete implementations of the ports defined in the Application Layer. This layer is also responsible for configuration management, initializing the appropriate adapters, and managing environment variables.

- **Controllers:** The Infrastructure Layer incorporates several REST [17] controllers that act as the primary entry points for external systems to interact with the application. These controllers expose application functionality via RESTful APIs[18], enabling external clients to trigger operations or query data.
- **Adapters:** Specific implementations of the application ports which act as a bridge between the core application logic and external systems or implementation.
- **MQTT Module:** The MQTT module handles message-based communication for the real-time data streams sent by the sensors. This component ensures reliable and scalable messaging, making it a crucial part of the infrastructure.
- **Persistence Module:** The Persistence module is responsible for managing data storage and retrieval operations by implementing the abstract repositories ensuring that all data interactions adhere to the database engine, guaranteeing performance and consistency requirements.
- **Config Module:** The Config module centralizes the configuration of the application, ensuring that runtime parameters, environment variables, and system-level settings are accessible and manageable in a unified manner. It is also responsible for loading the correct adapters for the ports.

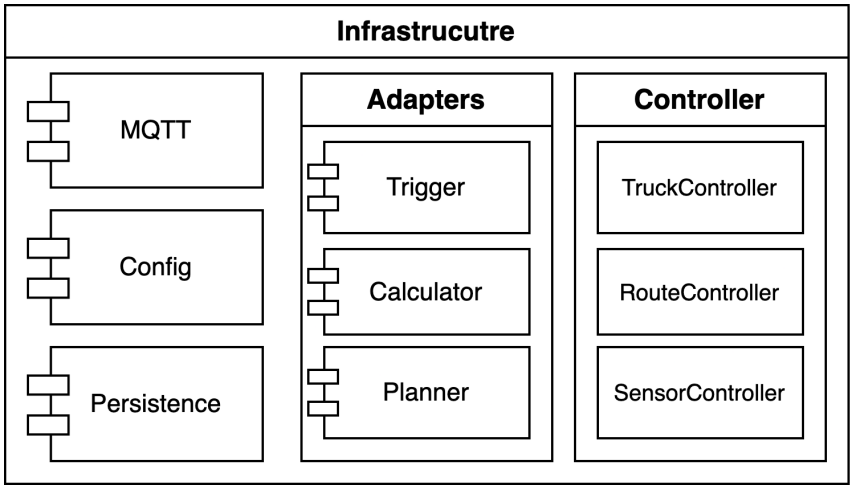

Figure 6.5: Overview of the Infrastructure Layer

Example Workflow: Adding a New Truck

1. **Inbound Request:** An HTTP `POST /trucks` call is received by `TruckController`, carrying truck details (e.g., plate number, depot location).
2. **Controller Handling:** The controller validates the request and creates a `TruckCreateRequest` object, passing it to `TruckService.createTruck(...)`.
3. **Application Service Logic:** The service applies domain rules, checks for conflicts (e.g., duplicate plates), and uses the `TruckRepository` port to persist the truck.
4. **Adapter Implementation:** In the infrastructure layer, the `TruckRepository` is implemented for the chosen database. It manages actual persistence details.
5. **Return Response:** If successful, the service returns a success object DTO, and the controller sends an HTTP `201 Created` status to the client.

6.3 Persistence and Data Handling

This section discusses the overall approach to storing and retrieving data in our application. A dual-database solution has been strategically chosen, using a *relational database* (PostgreSQL [19]) for the core entities and a *non-relational database* (MongoDB [20]) for handling route-related data. By taking advantage of each database's respective strengths, we can ensure consistency, ease of querying, and flexible geospatial capabilities.

Dual Database Strategy: Relational and Non-Relational

The system's persistent data is categorized into two primary domains:

- *Relational (PostgreSQL)*: Manages entities sensors and trucks, which benefit from a well-defined schema and transactional consistency.

- *Non-Relational (MongoDB)*: Stores route information in a format suited for geospatial operations and more fluid schema evolution.

Why Relational for Sensors and Trucks

We have chosen a relational database for the sensor and truck entities for several reasons:

- *Stable Schema Requirements*: The data model for sensors and trucks rarely undergoes drastic changes. A fixed table structure fits naturally.

- *Transaction Support*: *ACID* (atomicity, consistency, isolation, durability) transactions help ensure data integrity when updating a sensor's fill level or assigning trucks to new routes.

- *Clear Relationships*: Trucks and sensors require straightforward, well-defined relationships (e.g., fetching optimal truck). Relational schemas simplify this aspect.

Why Non-Relational for Routes

Routes can become much more complex due to their geospatial nature and potentially large amounts of path data. MongoDB's *document-oriented* model proves beneficial in:

- *GeoJSON [21] integration:* MongoDB natively supports storing and indexing data in the GeoJSON format, which enables geospatial queries (e.g., distance calculations, route visualizations).

- *Schema Flexibility:* Route structures can vary in length and detail (list of visited sensors, path geometry, etc.), making a document store more accommodating.

- *High Performance for Geospatial Operations:* As routes become longer or more complex, MongoDB's optimized queries for geospatial data help maintain responsiveness.

GeoJSON is a standard format for representing simple geographical features along with their non-spatial attributes. We make use of its feature types which include:

- `Point` — for single coordinates (e.g., sensor location, depot base of a truck).

- `LineString` — for paths (e.g., the truck's route through the city).

By storing an entire route as a *FeatureCollection* (containing a `LineString` for the route path and `Points` for sensor locations), the system can efficiently fetch and render a complete route in geospatial applications or queries.

6.4 MQTT and the Message-Driven Approach

This section describes how our application leverages the MQTT protocol to receive and process sensor data in near real time. MQTT adopts a *publish-subscribe* model, unlike traditional *request/response* paradigms such as *HTTP polling*. This model is well-suited for handling high volumes of asynchronous messages from sensors distributed across the city.

Message Flow

Figure 6.6 provides a conceptual view of how messages travel:

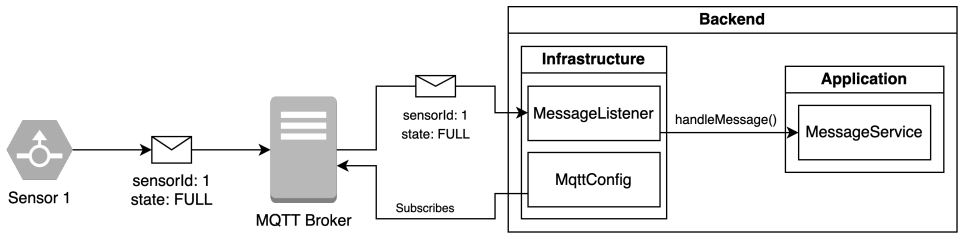

Figure 6.6: Sensor messages flow

1. *Sensor Publishes Update:* When a container's fill level changes (e.g., from *HALF* to *FULL*), the sensor device creates a message containing the sensor's unique identifier and the new fill state, then publishes it to the assigned topic.
2. *Broker Receives and Forwards:* The *MQTT broker* accepts the incoming message and forwards it to all subscribers listening on that topic.
3. *Application Consumes Message:* Our application, which has previously subscribed to the broker, processes the incoming payload. Internally, it validates or transforms the data and forwards it to the application layer service to handle the new message (e.g., to update the sensor's state in the database or trigger route-planning logic).

This publish-subscribe pattern allows us to easily scale to thousands of sensors. Each additional sensor starts publishing messages to the broker without necessitating changes to the core application logic are necessary, as it will automatically receive any newly published messages from subscribed topics.

Topics for Each Zone

In larger deployments with multiple city zones or a high volume of sensor data, it can be beneficial to subscribe only to the MQTT topics relevant to each specific zone. For instance, one instance of the application might subscribe to `sensors/zone1`, while another instance subscribes to `sensors/zone2`, and so on.

This conditional subscription strategy allows to scale out horizontally, distributing the

overall workload and improving responsiveness by ensuring that each application instance handles only the messages for its assigned zones.

For more details about scaling strategies, please refer to the discussion in 8.3.

Chapter 7

Implementation

This chapter presents the implementation details of our waste management system, *Higiea*, following the hexagonal (ports and adapters) architecture described in Chapter 6.

We focus on the core interfaces for route handling, the mechanisms for message-driven sensor updates, and the infrastructure-level adapters. Throughout, our goal is to illustrate how the architectural principles translate into concise and maintainable code while avoiding excessive detail that might obscure the key design decisions.

7.1 Java Spring Boot

Java Spring [22] is a widely used framework that simplifies the development of enterprise-level Java applications. By managing common concerns like object creation and data access, Spring promotes principles such as loose coupling and separation of concerns. Central to this is **dependency injection**, where Spring's Inversion of Control (IoC) container manages object creation and assembly, ensuring flexibility and modularity.

Spring organizes application components using *beans*—Spring-managed objects similar to *singletons* and available in the application's context, allowing them to be easily injected and used where needed. These are annotated for clarity:

- `@Component`: General-purpose bean.
- `@Service`: For business logic.
- `@Repository`: For persistence logic.
- `@Configuration`: For application setup.

This framework ensures clear separation of concerns, enabling independent development, testing, and maintenance of application components, such as domain logic and external integrations.

We have utilized Spring for the implementation of Higiea, leveraging its modular architecture to ensure that the domain logic (e.g., sensor and route services) and external integrations (e.g., MQTT clients, Azure Maps adapters) remain decoupled while seamlessly integrated.

7.2 Message Handling and Business Logic

Sensor messages arrive through MQTT and are processed by our `MessageService`, a class belonging to the application layer. As shown conceptually in Figure 7.1, this service coordinates with domain services—such as `SensorService`, `TruckService`, and `RouteService`—while relying to route-related ports when needed.

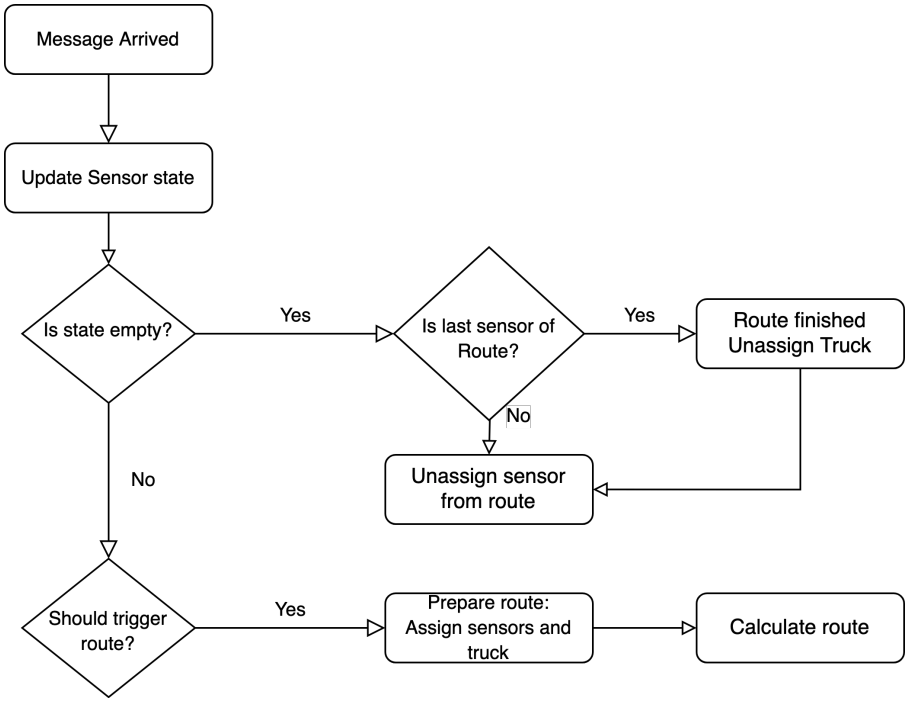

Figure 7.1: Conceptual message handling flow in `MessageService`

1. **State Update:** The `sensorService.updateSensorState(...)` call updates the sensor in the domain and database.
2. **Route Trigger Check:** If the sensor is not empty, the `RouteTrigger` port decides whether to initiate a new route.
3. **Route Planning:** If a route is triggered, `RoutePlanning` chooses an optimal truck and a set of candidate sensors.
4. **Route Calculation:** Finally, `RouteService` interacts with the `RouteCalculator` port to compute and store the final route path for dispatch and saves it.

7.3 Route Ports

An essential feature of *Higiea* involves automatic route determination based on sensor updates. The system must decide *when* to trigger a new route, *which* sensors should be included, and *how* to calculate the most efficient travel path.

To support these concerns, we define three primary *ports* (`RouteTrigger`, `RoutePlanning`, and `RouteCalculator`)

RouteTrigger

The `RouteTrigger` port (Listing 7.1) returns a `Mono<Boolean>` indicating if a new route should be triggered. It can be adapted for different rule sets to determine when a route should be triggered such as upon first truck availability or like in our case, when a trash containers has reached full capacity [7.4].

```
package com.ub.higiea.application.ports;

public interface RouteTrigger {
    Mono<Boolean> shouldTriggerRoute(Sensor sensor);
}
```

Listing 7.1: Excerpt of the `RouteTrigger` port

RoutePlanning

The `RoutePlanning` port is responsible for constructing a route configuration, returning a list of sensors whose trash containers need to be emptied by a truck. It can also be adapted to follow different strategies to include different trucks and sensors.

```
package com.ub.higiea.application.ports;

public interface RoutePlanning {
    Mono<Tuple2<Truck, List<Sensor>>> prepareRoute();
}
```

Listing 7.2: `RoutePlanning` port for selecting trucks and sensors

RouteCalculator

The `RouteCalculator` port encapsulates external route computation logic. For instance, we can integrate with Azure Maps or any other mapping provider. Listing 7.3 and Listing 7.4 illustrate the contract.

```
package com.ub.higiea.application.ports;

public interface RouteCalculator {
    Mono<RouteCalculationResult> calculateRoute(Location depotBase,
        List<Sensor> sensors);
}
```

Listing 7.3: `RouteCalculator` port for path computation

```
package com.ub.higiea.application.ports;

public record RouteCalculationResult(
    List<Sensor> orderedSensors,
    Double totalDistance,
    Long estimatedTimeInSeconds,
    List<Location> routeGeometry
) {}
```

Listing 7.4: `RouteCalculationResult` DTO holding the route outcome

7.4 Infrastructure Adapters

In the infrastructure layer, we implement the defined *ports* as specific classes. These adapters interact with external systems or apply custom algorithms, consistent with the *Ports and Adapters* approach.

RouteTrigger Adapter

To fulfill the objective of ensuring that no container remains full for an extended period, the `RouteTrigger` adapter initiates a route when a sensor transitions to a *FULL* state(Listing 7.5).

The underlying premise is straightforward and aligned with our application requirements: whenever a container becomes full, it should be collected as soon as possible if it has not yet been assigned to a route.

This approach reduces the risk of overflowing containers and improves the timeliness of garbage collection.

```
package com.ub.higiea.infrastructure.adapters.route.trigger;

@Component
public class FullContainerRouteTrigger implements RouteTrigger {
    @Override
    public Mono<Boolean> shouldTriggerRoute(Sensor sensor) {
```

```
                return Mono.just(sensor.getContainerState() ==
                    ContainerState.FULL);
            }
        }
```

Listing 7.5: Example `FullContainerRouteTrigger` adapter

RoutePlanning Adapter

The `RoutePlanning` adapter addresses the question of *which* containers (i.e., sensors) and *which* truck(s) should handle the new route. Our planning logic takes the following approach:

1. **Priority on FULL & HALF-FULL Sensors:** Only route-unassigned sensors with full or half-full states are considered.

2. **Optimal Truck Selection:** The algorithm attempts to find a truck with sufficient load capacity to collect all relevant containers in one pass. If it fails to locate such a truck (e.g., an especially large load or limited truck capacities), it falls back to the highest-capacity truck and reduces the sensor list until the total fill volume fits.

This planning logic delivers two main benefits. First, it guarantees that container fill states are addressed proactively, aligning with the objective of minimizing container overflows. Second, it preserves operational efficiency by sending the optimal truck per triggered event whenever feasible.

RouteCalculator Adapters

We have implemented the port `RouteCalculator` integrating *Azure Maps* [23] service to compute the most efficient travel path (Listing **??**). This choice directly supports the system's *production-readiness* by leveraging a commercially supported, globally available mapping service.

Azure Maps provides the following advantages:

- *Reliable Global Coverage:* Cities worldwide can use the same route-calculation service, ensuring consistency even as the application scales to multiple regions.

- *Traffic-Aware Routing:* Azure Maps supports real-time traffic data and historical trends, paving the way for future improvements in route optimization and predictive scheduling.

- *Asynchronous Clients:* The *MapsRouteAsyncClient* (from the Azure Maps SDK) aligns with our Reactive Programming approach, ensuring non-blocking calls and streamlined concurrency handling within Spring WebFlux (Spring's reactive-stack web framework).

7.5 High-Level Data Access Pattern

Reflecting the *Hexagonal Architecture* principles, the implemented data access pattern ensures domain models remain independent of persistence technology. We divide responsibilities into:

- **Domain Models:** These plain classes capture business rules (e.g., a truck's capacity constraints) without any reference to database logic.
- **Repository Interfaces:** Defined at the domain layer to outline common operations such as *find by ID* or *save*.
- **Infrastructure Entities:** Classes that map to database schemas or collections. They contain the necessary annotations and structural definitions for each persistence technology.
- **Mappers:** Utility components in the infrastructure layer. They translate domain models to infrastructure entities (e.g., valid PostgreSQL entity or MongoDB document) and translate back to domain models when retrieving data.
- **Repository Implementations:** Residing in the infrastructure layer, they rely on the appropriate database drivers or frameworks to execute queries. These implementations delegate object conversion to the mappers, ensuring the rest of the application remains database-agnostic.

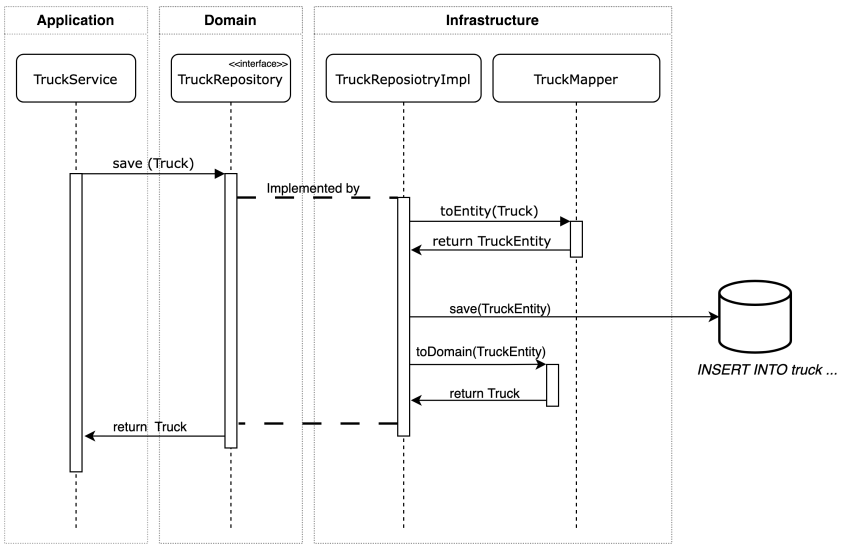

Figure 7.2: High-Level Flow of Data Through Service, Domain, Mappers, and Infrastructure Entities

Figure 7.2 illustrates the sequence of interactions across the layers when saving a Truck:

1. The TruckService in the application layer calls a domain repository interface to save a domain object.
2. The infrastructure-layer repository adapter implementation receives the request and uses a mapper to convert the domain model into the corresponding entity class.
3. The entity class is then persisted using the *relational* operation.
4. For reads, the flow is reversed: the implementation fetches infrastructure entities from the database, uses a mapper to transform them into domain models, and returns them to the application.

Advantages of the Approach

This abstraction allows us to swap in new database engines or alter data structures without changing business logic. The separation of concerns also aids in testing and maintenance, as domain behaviors can be tested purely in memory by mocking repository interfaces, thus isolating the domain logic from the actual database layer.

The approach provides several advantages:

- *Clear Separation of Concerns:* Domain logic, database specifics, and data mapping remain logically separated, enhancing maintainability.
- *Scalability and Evolution:* Each storage layer can be tailored independently as the application evolves (e.g., introducing advanced geospatial indexes in MongoDB or partial indexes in PostgreSQL) without requiring major refactoring of the core domain.
- *Ease of Database Migration:* Switching databases or modifying existing schemas is simplified, as changes are isolated to the infrastructure layer, minimizing disruption to the overall application.
- *Clean and Clear Structure:* As the application grows, this approach ensures that the system remains organized, making it easier to manipulate, extend, or debug when necessary.

7.6 Testing

A robust testing strategy is crucial for maintaining code quality and ensuring reliable functionality. To keep our application continuously validated and to ensure that new changes do not break existing functionality, we have adopted three complementary test strategies:

Unit Testing

Unit tests verify individual components. These tests focus on internal logic (e.g., sensor or truck creation, route calculations) by mocking external dependencies (repositories or other services). We have implemented this type of tests for our most important

classes such as the ones in the modules `Services`, `DomainServices`, `RoutePlanning` and `Controllers`.

This approach isolates each class and method and confirms correct behavior under different scenarios and edge cases.

Integration Testing

Integration tests validate interactions among various modules and internal workflows. They also verify infrastructure layers such as database or messaging components.

Controller endpoint tests confirm correct request processing, domain services are invoked properly, and responses (HTTP status codes, JSON payloads, error messages) match expectations. For instance, they ensure a missing resource leads to `404 Not Found`.

By covering these interactions, integration tests preserve overall system quality and confirm that *Higiea*'s core logic remains consistent even as features evolve.

End-to-end Testing

End-to-end (E2E) tests simulate real-world conditions in an environment closely mirroring production.

We have built a simulator that publishes MQTT messages to the cloud-deployed broker, allowing us to confirm that our backend accurately listens, receives, and processes these messages. Additionally, the simulator visualizes sensor updates, newly created routes, and their assignments.

For more details on the simulation setup, refer to Chapter 9.1.

Chapter Summary

This chapter has detailed the *hexagonal architecture* implementation in Higiea, including key ports (`RouteTrigger`, `RoutePlanning`, `RouteCalculator`), infrastructure adapters for external integrations (Azure Maps, MQTT), and repository patterns that isolate the domain model from persistence concerns. The testing strategy—covering both unit and integration tests—ensures each layer's correctness and the overall robustness of the system.

Chapter 8

Cloud Infrastructure and Scalability

This chapter explores the foundational elements of Higiea's cloud infrastructure and the strategies implemented to ensure *scalability* and *high performance* across diverse deployment scenarios.

We begin by highlighting the platform's essential components, including our application running on the cloud, managed database services for both relational and geospatial data, and a robust MQTT broker for sensor communication. From there, we delve into the security measures that safeguard data at rest and in transit, ensuring that as Higiea scales, it maintains the reliability and protection that waste management operations demand.

Finally, we examine how *containerization*, *horizontal scaling*, and *zone partitioning* collectively enable Higiea to adapt seamlessly—from small towns with minimal sensor activity to sprawling urban centers requiring extensive coordination.

8.1 Component Overview

In this section, we present the cloud-based deployment strategy for our Higiea platform, highlighting the services used, their benefits, and security measures. As illustrated in Figure 8.1, the infrastructure comprises the following components:

- **Application Container:** Docker Image [24] uploaded to *Azure Container Registry* [25] and deployed on *Azure Container Apps [26]*.
- **Databases:** *Azure Database for PostgreSQL Flexible Server [27]* for relational data, and *Azure Cosmos DB for MongoDB [28]* for geospatial and document-based data.
- **MQTT Broker:** Deployed on *EMQX* [29].
- **External Service Adapter:** Azure Maps [23] integration for route calculation.

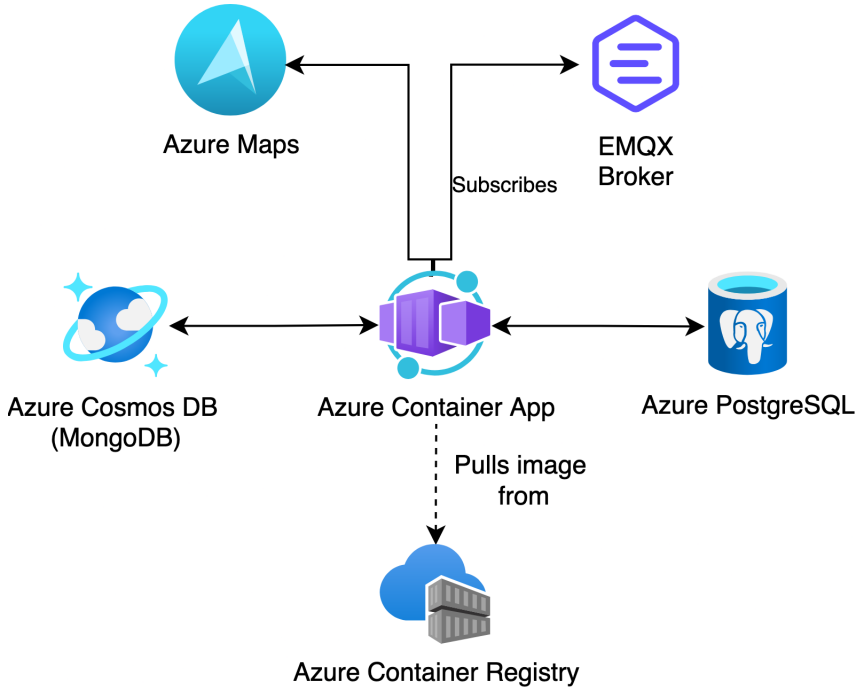

Figure 8.1: High-Level Cloud Infrastructure and Deployment Model

We will describe each component in detail, explaining the rationale behind their selection, the benefits of adopting these services, and the corresponding security mechanisms to protect data and services.

Docker for Containerization

Docker is an open-source platform that simplifies building, deploying, and running applications using *containers*. A Docker *image* serves as a lightweight, immutable scheme containing the application, its dependencies, and runtime environment. From these *images*, *containers* are created as isolated, portable runtime instances of the application.

- *Consistency and Portability*: Docker containers encapsulate the application and its dependencies, ensuring the same version of libraries, frameworks, and runtimes can be deployed consistently on any host system that supports Docker.
- *Resource Efficiency*: Docker containers are more lightweight compared to traditional virtual machines, resulting in faster startup times and reduced resource utilization.
- *Scalability*: Rapid creation and destruction of containers enable horizontal scaling based on demand.

Within Higiea, our application has been packaged into a Docker image. Containerizing the backend guarantees a reproducible environment for development, testing, and production scenarios.

Azure Container Registry and Container Apps

Azure Container Registry (ACR) is a managed service by Microsoft Azure that acts as a private repository for Docker images. It allows us to store, manage, and version Docker container images efficiently.

When a new release is triggered, the CD workflow, orchestrated by GitHub Actions, is activated. This workflow automates the process of building the Docker image for the updated application. Once built, the image is pushed to ACR, making it readily available for deployment.

Once the Docker image is uploaded to ACR, our application is deployed to *Azure Container Apps*. This service provides a *serverless* platform, meaning the underlying infrastructure is automatically managed by Azure. Removing this way the need to manually handle servers or virtual machines. This ensures resources are optimized and costs are minimized while maintaining performance and deploying new instances, scaling to meet demand, and maintaining the system's performance become seamless

The deployment workflow is illustrated in Figure 8.2, where the process begins with the GitHub Actions-based CD pipeline and culminates in the deployment of containerized applications in Azure Container Apps.

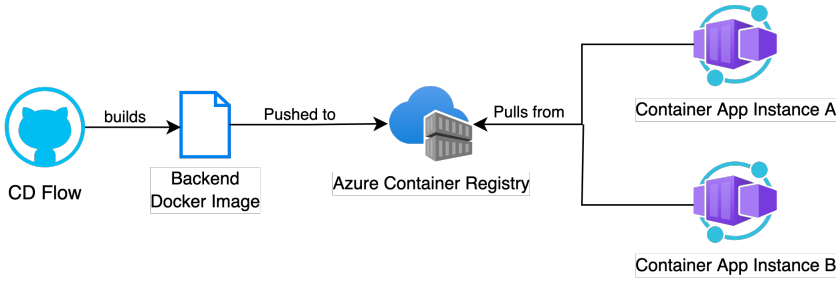

Figure 8.2: Deployment Workflow

Azure PostgreSQL and Azure Cosmos DB

For managing our application's data needs, we employ a combination of *Azure Database for PostgreSQL Flexible Server* and *Azure Cosmos DB* for MongoDB.

- *Azure PostgreSQL Flexible Server*: Fully managed, with automated patching, high availability, and flexible scaling options.
- *Azure Cosmos DB (MongoDB)*: Handles route data via its MongoDB-compatible API, offering global distribution, automatic indexing for geospatial queries, and elastic scalability.

These database services together offer a robust and flexible foundation for data management.

MQTT Broker Deployment with EMQX

We deploy *EMQX*, a highly scalable and distributed MQTT broker in the cloud. *EMQX* efficiently handles large volumes of concurrent sensor connections and ensures fault tolerance through clustering capabilities. It also provides robust security measures, such as *Transport Layer Security (TLS)[30]/Secure Sockets Layer(SSL)[31]* encryption, role-based access control, and authentication mechanisms.

This setup ensures reliable and secure message routing between sensors and our backend, supporting seamless communication even as the number of sensor connections increases or new topics are introduced.

Integration with Azure Maps for Route Calculation

For route calculation, we utilize *Azure Maps* through the `AzureMapsRouteCalculatorImpl` adapter to calculate optimal routes in real time. This service is crucial for determining the shortest path for trucks to traverse all specified waypoints, including sensor locations, and return to their starting point. *Azure Maps* provides advanced geospatial capabilities, integrating real-time traffic data, historical trends, and wayfinding algorithms to adapt dynamically to changing conditions.

The integration ensures precise routing by calculating the most efficient sequence to go though all sensor waypoints, minimizing travel distance and time while considering live traffic conditions. By leveraging *Azure Maps* global coverage and robust geospatial APIs, Higiea delivers highly responsive route calculations, enhancing the system's ability to meet operational demands.

In alignment with the *Ports and Adapters* architecture, the `RouteCalculator` port can be adapted by an alternative provider or custom implementation if additional routing or geospatial requirements emerge. This modular approach not only facilitates seamless interaction with the external service but also ensures scalability for future needs.

8.2 Cloud Security and Communication Channels

Ensuring security across all communications and data storage points in our system is paramount. Our deployment on Azure follows best practices to protect data in transit and at rest:

- *Username and Password Protection*: Services PostgreSQL, Cosmos DB and EMQX requires valid credentials to authenticate and access resources.
- *Encrypted MQTT Communication: WebSocket Secure* (WSS) is an encrypted version of WebSocket communication, relying on SSL/TLS protocols to ensure secure data exchange over the network. By using WSS, messages from sensors to the broker are encrypted, preventing unauthorized interception or tampering during transmission.

- *Database SSL/TLS:* SSL and its successor, TLS, are cryptographic protocols designed to provide secure communication over a network. For PostgreSQL and Cosmos DB, SSL/TLS ensures that all data transmitted between the database and the client is encrypted. This encryption protects against data interception, eavesdropping, and tampering by malicious actors.
- *Azure Maps Data Protection:* Requests to Azure Maps are secured using Azure's identity and access management. Only authorized services or principals can invoke the route APIs, ensuring secure interaction with mapping services.
- *Azure Container Registry Security:* ACR leverages *Azure Active Directory* (Azure AD) integration for authentication and authorization, ensuring that only approved users or services can push or pull container images. Additionally, network security features like private endpoints can restrict registry access to trusted networks.
- *Azure Container Apps Protection:* Azure Container Apps benefit from *Azure-managed identity and role-based access control* (RBAC), which secures interactions with other Azure resources. Traffic to deployed applications can be secured using Azure's built-in TLS encryption, while network isolation through virtual networks ensures that applications remain protected from unauthorized access.

Overall, this approach to cloud infrastructure and deployment ensures that our application is scalable, fault-tolerant, and secure. By leveraging containerization via Docker, managed database services, a robust MQTT broker, and a proven geospatial service (Azure Maps), Higiea can reliably scale to meet dynamic workloads while maintaining stringent security standards.

8.3 Scalability and Performance Considerations

One of the key non-functional requirements for Higiea is *NFR5: Adaptability*, which demands that the system operate efficiently across varying deployment environments. These can range from small towns with just a few sensors and a single truck to large metropolitan areas with thousands of sensors and multiple truck fleets.

Maintaining performance and stability under these diverse conditions is essential to ensure reliable waste collection and resource optimization.

Scaling Strategies

Vertical scaling (also known as *scaling up*) involves increasing the computational resources of a single machine or instance. For example, you can add more CPU cores, more RAM, or faster storage.

While this approach is straightforward, it has inherent limitations, including physical

resource caps and the risk of creating a single point of failure, leaving the system vulnerable to crashes from unexpected workload surges or hardware failures.

Horizontal scaling (also called *scaling out*) focuses on adding more instances or machines, each handling a portion of the system load. Rather than making a single server more powerful, you run multiple, smaller servers or containers in parallel, distributing the workload among them. The benefits include:

- *Elasticity*: Easier to increment or decrement capacity by adding or removing additional nodes (e.g., Docker containers or VMs).
- *Reliability and Resilience*: The failure of a single instance has less impact as requests can fail over to other healthy instances.
- *Cost-Effectiveness*: Typically cheaper and more flexible to add more commodity servers or containers than to invest in a single high-end machine.

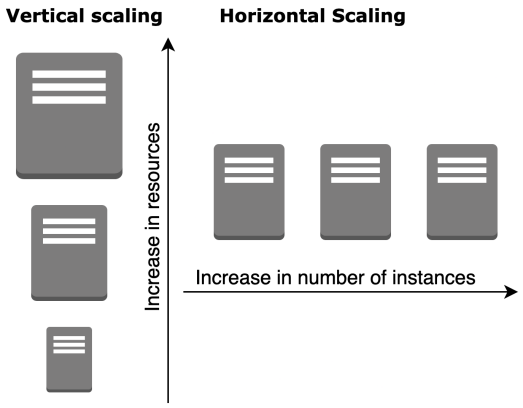

Figure 8.3: Different scaling strategies

Given Higiea's requirement for flexibility and resilience, horizontal scaling is particularly well-suited. It ensures that as sensor data grows, the system can adapt by distributing the load across multiple instances, thereby offering robust performance and preventing bottlenecks.

It is important to note that both strategies can and often should be used in tandem. However, Higiea has been specifically designed to achieve its full potential by prioritizing the horizontal scaling strategy, as will be explained below.

Zone Partitioning and Message Handling

A fundamental strategy for Higiea's horizontal scaling is *zone partitioning*, where each container instance is assigned to a specific geographic region or set of garbage containers. This approach aligns with real-world logistics, as garbage trucks typically operate within

defined zones and do not traverse the entire city just to empty one container.

Higiea leverages MQTT's publish-subscribe model to separate sensor data by zones. Each instance subscribes only to the MQTT topics relevant to its designated zone or subset of sensors. This localizes sensor traffic and simplifies concurrency because each container processes a smaller, more coherent data set, reducing both complexity and potential message-processing delays.

Horizontal Scaling with Multiple Container Apps

To enable horizontal scaling, Higiea relies on Azure Container Apps to deploy multiple instances of the same Docker image stored in Azure Container Registry. Each instance of the Higiea backend is configured at runtime via *environment variables* that specify:

- *Database Endpoints*: URLs, credentials, and ports for PostgreSQL and CosmosDB.
- *External Services*: API keys and endpoints for Azure Maps.
- *MQTT Topic or Zone*: Indicates which topic or city zone the instance subscribes to from the EMQX broker.

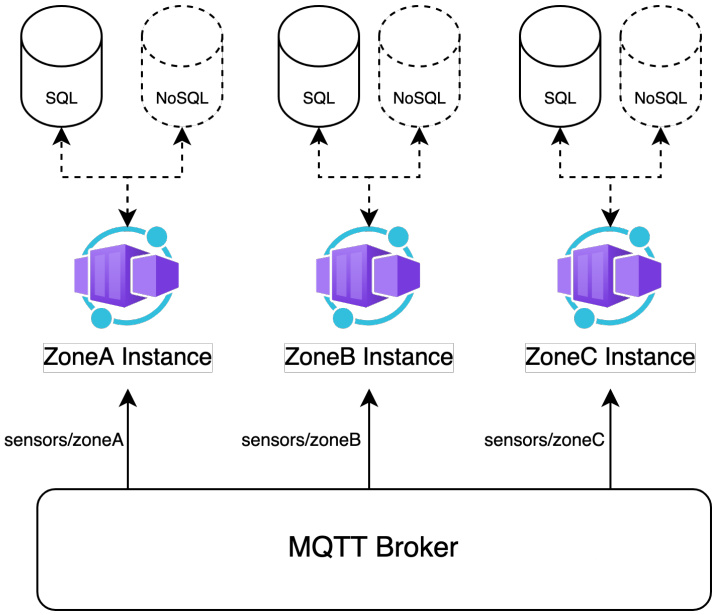

Figure 8.4: Multi-instance architecture for Higiea, illustrating zone-specific container deployments.

As shown in Figure 8.4, separate Azure Container Apps may be deployed for *ZoneA*, *ZoneB*, and *ZoneC*. Each container only processes and stores data relevant to its assigned zone.

This design simplifies data management and ensures that performance remains predictable, even as the number of sensors grows.

Concurrency and Asynchronous Messaging

Higiea's sensor network heavily utilizes MQTT's publish-subscribe mechanism, which is inherently *asynchronous*. If a large number of sensors publish simultaneously to the same topic, a single server—even if scaled vertically—could face *concurrency bottlenecks* or *race conditions*.

However, when distributing sensor traffic across multiple horizontally scaled instances effectively mitigates these issues:

- Each instance manages a smaller number of incoming messages, reducing queue times and processing overhead.
- Potential overloads in one instance do not affect other instances, improving overall system resilience.

By spreading the load across multiple container apps, Higiea also ensures that unexpected traffic surges, sensor malfunctions, or network rerouting do not overwhelm a single point in the architecture.

Deployment Scenario Adaptability

Small Town Deployments: For communities with only a handful of sensors and a single truck, Higiea can run on a single container instance. The resource usage remains minimal, and the overhead is low, meeting the system's requirements with ease.

Metropolitan Deployments: In larger cities, the number of containers (each representing a truck or zone) can be scaled as needed. Each instance manages a different zone or subset of zones, ensuring that no single container is overloaded. This makes performance predictable and efficient, even under intense workloads.

In summary, horizontal scaling coupled with zone partitioning ensures that Higiea remains highly adaptable across different deployment scales.

By splitting the workload and leveraging containerization, the system can handle everything from low-volume sensor environments to sprawling metropolitan data traffic with reliability and cost-effectiveness.

Chapter 9

Evaluation and Results

In this chapter, we present a comprehensive evaluation of the Higiea platform's functionalities, performance, and practical feasibility.

We begin by describing a web-based simulator that replicates real-world conditions—covering sensor placement, message-driven updates, and multi-zone deployments—to demonstrate end-to-end system behavior.

We then delve into *monitoring* and *logging* strategies that ensure reliability and maintainability, and wrap up with a cost analysis highlighting key financial considerations for production use.

Through these sections, we illustrate how Higiea meets its non-functional requirements for scalability, adaptability, and performance, while maintaining a user-friendly interface for operators and city administrators.

9.1 Simulator

To validate and visualize the system's behavior under realistic conditions, a web-based simulator has been developed. This simulator functions as a comprehensive testbed that reproduces key aspects of a real-world deployment, enabling end-to-end testing and performance analysis. Specifically, it helps to:

1. **MQTT Broker Validation:** Confirming that the MQTT broker reliably receives and forwards sensor state messages, reflecting typical IoT communication patterns.
2. *Backend Message Processing:* Verify that the Higiea backend accurately interprets sensor updates, applies domain logic (e.g., RouteTrigger and RoutePlanning), and maintains data consistency.
3. **Service Integrity and Functionality:** Ensure reliable interactions among all services, including event triggering, route planning, and capacity-based allocation of sensors to routes.

4. **API REST Endpoints:** Test that RESTful endpoints handle client requests correctly (e.g., placing sensors, adding trucks, or terminating routes).
5. **End-to-End Testing:** Demonstrate the full workflow: sensor and truck creation and state updates →→ real-time messages via MQTT →→ route calculation →→ final route visualization.
6. **Multi-Instance Scenarios:** Assess the system's capacity to run multiple service instances (zones) concurrently, each with its own REST/MQTT endpoints, without conflict or inconsistency.

User Interface

Figure 9.1 displays the main interface of the simulator for a single zone, highlighting the major functionalities:

- *Zone Selection:* Operators can switch among different zones (e.g., distinct neighborhoods) using a dropdown menu.
- *Entity Placement:* Buttons for adding new *Sensors* or *Trucks* within the selected zone.
- *Sensor Monitoring:* The map renders the truck's depot location in blue and each sensor location in color-coded states (`EMPTY` =green, `HALF`=orange, `FULL`= red). Enabling real-time monitoring of sensor statuses and covering UC1.
- *Truck Route Selection:* A dropdown for available trucks shows their assigned routes, enabling operators to inspect each path individually.

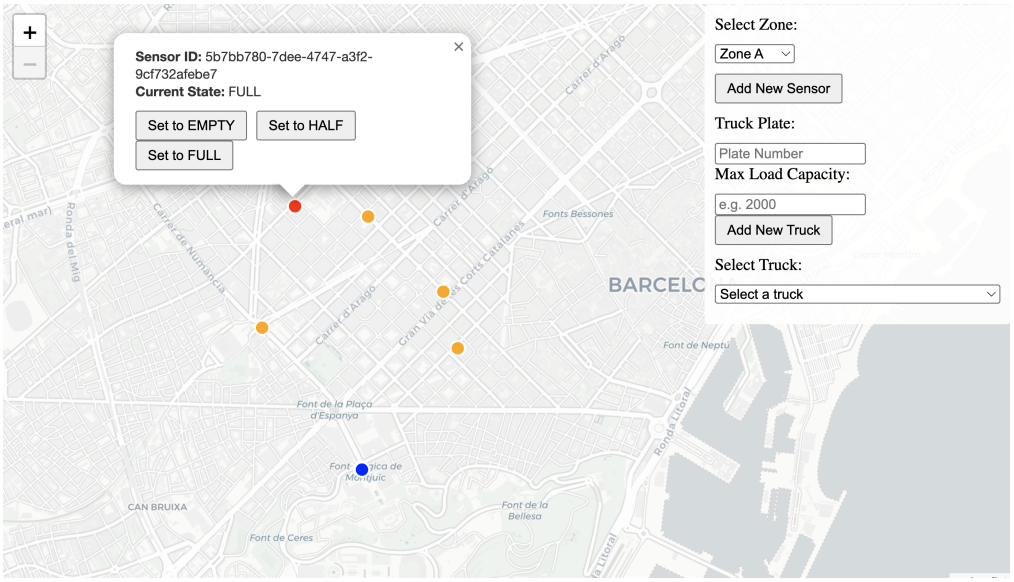

Figure 9.1: Main view of the simulator for Zone A, showing the user interface, zone selection options, and tools for sensor/truck placement and route inspection.

When the user clicks a sensor icon, a small panel indicates the sensor's unique ID, current state, and potential actions to update its status. For a truck, the panel shows the truck's plate number, capacity, and a control to send a "route-termination".

Placing Sensors and Trucks

The simulator allows operators to create sensors and trucks in near real time, replicating the deployment of IoT devices in a city environment. Once a placement event occurs:

- *Sensor Placement:* Clicking on the map at the desired location issues a `POST` request to the Higiea backend, creating a new sensor with default state `EMPTY`. In production-like tests, this request completes in around 100ms, reflecting the system's capacity for *NFR1 (Performance)* in terms of rapid sensor registration. The newly added sensor appears on the simulator's map in *green*, denoting an empty state.
- **Truck Placement:** Similarly, adding a truck requires specifying a *maximum load capacity* and *plate number*, then clicking on the map to post the truck's location. A `POST` request registers the truck in the backend (e.g., `Truck 0250AAA Capacity: 250`). The truck's depot is displayed in *blue*, providing a clear visual anchor for route calculations.

To keep the interface uncluttered, the simulator filters `GET` requests to retrieve only sensors and trucks relevant to the currently selected zone. This approach promotes scalability, allowing large cities to be split into multiple zones, each with its own dedicated map view.

Updating Sensor States via MQTT

After creation, each sensor can switch among three states: `EMPTY` (green), `HALF-FULL` (orange), or `FULL` (red), reflecting UC2. Changing a sensor's state in the simulator replicates how a real IoT sensor might detect and broadcast fill-level changes:

Rather than sending an HTTP request, the simulator publishes an MQTT message to the topic specific zone. This closely mirrors actual sensor devices sending status updates to the broker.

The map icon updates in real time, visually reflecting state transitions (e.g., from half-full to full).

Automatic Route Triggering

Figure 9.2 shows the simulator immediately after a sensor transitions to `FULL`, addressing UC3. The system's `RouteTrigger` port evaluates whether a new route is required and, if so, invokes `RoutePlanning` to select a truck and the relevant sensors and calculat the optimal path.

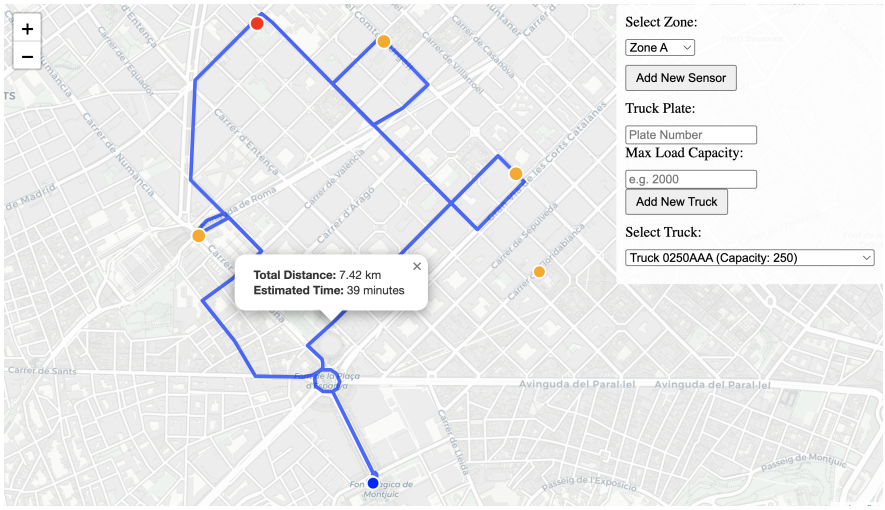

Figure 9.2: Example route after a sensor has changed to `FULL`; a blue line indicates the truck's path.

In the illustrative example:

- *Capacity Heuristic:* A `FULL` sensor is given a "capacity cost" of 80, whereas `HALF-FULL` sensors cost 50. This verifies (UC4) —*Route Planning*—which determines whether all sensors fit in a single truck.
- *Truck Assignment:* The truck with capacity 250 can pick up one full sensor and up to three half-full sensors, leaving any additional half-full sensor(s) unassigned if the total load would exceed capacity.
- *Route Visualization:* The `RouteCalculator` (see UC5) integrates with Azure Maps to plot a path from the truck's depot, visiting assigned sensors in an optimal order, then returning to the depot.

It is important to note that the message publishing from the sensor side and the pulling in the backend does not involve an HTTP request waiting for a response. As a result, to see the created route, it is necessary to wait a few seconds and refresh the simulator.

Clicking the route on the simulator reveals the estimated driving time and distance considering real time traffic. This feature is particularly useful for testing the correctness of the integration with Azure Maps and verifying that real-time route computations are accurate.

Multiple Zones and Multi-Instance Deployments

In keeping with *NFR1 (Scalability)* and *NFR6 (Adaptability)*, Figure 9.3 illustrates the scenario when running multiple instances, in this case responsible for *Zone B*, with its own MQTT topic and REST endpoints:

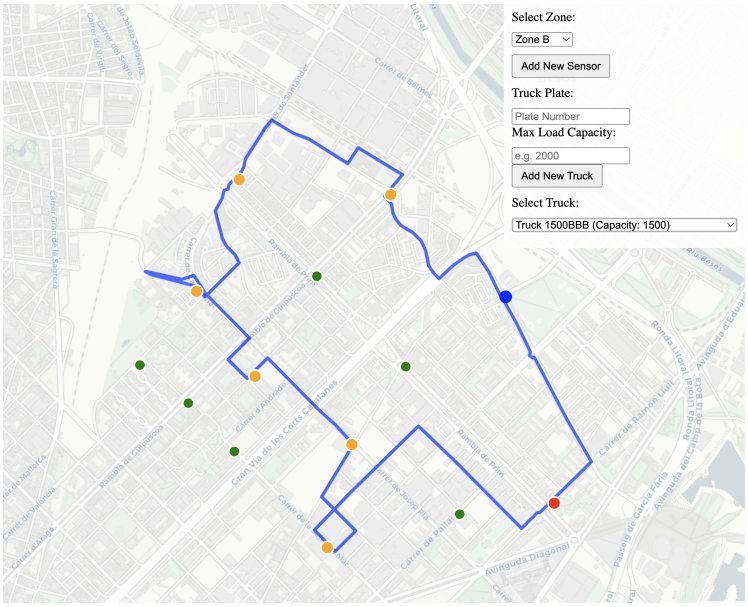

Figure 9.3: Alternate simulator view for Zone B. A separate instance handles all sensors and trucks for this area.

Treating each zone as a separate service instance makes scalability more manageable, fulfilling *NFR3 (Modularity)* and *NFR4 (Maintainability)*,; each backend focuses on its local data, preventing performance bottlenecks and ensuring consistent state updates across different neighborhoods.

9.2 Logging and Monitoring

A robust logging and monitoring strategy is vital for ensuring *NFR5 (Reliability)* and *NFR4 (Maintainability)* of the system, as well as for enabling operators to investigate performance, diagnose errors, and optimize resource usage.

Higiea captures logs for every major event, including sensor state changes and automatic route triggering, and stores these logs in a central repository. The log excerpt below illustrates the moment a sensor changes its state to FULL in Figure 9.2, triggering a new route and assigning the sensors and truck:

```
2025-01-07T18:54:39{Europe/Madrid} INFO com.ub.higiea.application.services.MessageService-
    Handling message for sensor ID: 16d39cca-5655-4260-85ba-33c682c07faf, state: 80

2025-01-07T18:54:39{Europe/Madrid} INFO com.ub.higiea.application.services.MessageService-
    Route trigger condition met. Triggering route.

2025-01-07T18:54:39{Europe/Madrid} INFO com.ub.higiea.application.services.MessageService-
    Sensors assigned to the route: [...]
```

```
2025-01-07T18:54:39{Europe/Madrid} INFO com.ub.higiea.application.services.MessageService-
    Preparing route with truck ID: 26c98621-20fd-4a10-80ac-f6fa784cb438

2025-01-07T18:54:40{Europe/Madrid} INFO com.ub.higiea.application.services.MessageService-
    Route preparation completed successfully.
```

Each line includes precise timestamps in local server time, facilitating chronological event reconstruction and correlation with sensor activity.

In addition, different log levels can be assigned based on the nature and severity of the event being logged (`DEBUG`, `TRACE`, `INFO`, `WARNING`, `ERROR`).

Integration with Azure Log Analytics

To facilitate long-term storage, advanced search, and data visualization, these logs are streamed into an *Azure Log Analytics Workspace* [32]. Operators can:

- *Perform Query-Based Analysis* to filter and aggregate events (e.g., by sensor ID, truck ID, or time range).
- *Visualize Key Metrics* to track the frequency of route triggers, average sensor fill times, or truck utilization.
- *Generate Alerts* for anomalous behaviors, such as an unexpected surge in FULL states or a repeated route-planning failure.

9.3 Cost Analysis

This section estimates the economic costs for deploying and maintaining the Higiea platform, focusing on two main areas: *cloud services* and *human resources*. These figures provide a high-level sense of monthly and annual budget requirements based on publicly available pricing information, but actual costs vary depending on usage patterns and negotiated enterprise discounts.

Context

For illustrative purposes, we consider two potential deployment sizes:

- *Small-Town Pilot:* Population around 50,000. We assume a limited number of sensors (e.g., 200–500) and 1–2 garbage trucks.
- *Moderate-Scale City:* Population ranging from 200,000 to 500,000, with 2,000–3,000 sensors and a larger fleet of trucks (5–10).

These population sizes are indicative of typical small or mid-sized European towns (50,000 inhabitants) vs. a larger urban center. The total cost is driven chiefly by three factors:

1. *Sensor count and frequency of updates,* impacting MQTT and database throughput.
2. *Number of daily route calculations,* affecting Azure Maps usage.
3. *Level of redundancy,* scaling, and uptime requirements for critical services.

Developer Salaries: A typical small team of 2–3 software developers and one DevOps engineer incurs an estimated monthly cost of €12,000–€16,000 in many European markets.

Table 9.1: Indicative Monthly Azure Service Costs (in EUR) for Two Deployment Scales

Service	**Small-Town Pilot**	**Moderate-Scale**
Compute and Registry		
Azure Container Registry	€5–€10	€5–€15
Azure Container Apps	€60–€110	€180–€370
Databases		
Azure PostgreSQL	€50–€90	€180–€370
Azure MongoDB	€30–€60	€280–€550
Messaging and Location		
EMQX	€40–€75	€90–€140
Azure Maps	€10–€30	€50–€150
Estimated Monthly Costs	**€195–€375**	**€785–€1,595**

In conclusion, an initial deployment of Higiea for a town with 50,000 inhabitants requires minimal infrastructure and costs approximately €220/month for essential cloud services, excluding staffing.

More extensive setups handling thousands of sensors scale proportionally in both resource usage and costs, with developer salaries as the dominant ongoing expense.

Summary

Overall, this chapter verifies that Higiea performs as intended under a variety of operational scenarios and provides a cost-effective roadmap for real-world deployment. With the essential evaluation and results in place, the subsequent chapter will draw final conclusions from these findings and propose avenues for future enhancements, ensuring that Higiea remains well-positioned to evolve alongside emerging urban waste management challenges.

Chapter 10

Conclusions and Future Work

The primary objective of this project was to propose and implement a waste management platform—*Higiea*—that leverages Internet of Things (IoT) technology, automated route planning, and scalable cloud infrastructure to address the inefficiencies of traditional collection methods. This concluding chapter summarizes the work done, offers a critical assessment of the solution's advantages and limitations, and proposes directions for continued development.

Summary of the Work Carried Out

The work began by examining the current challenges in urban waste management, highlighting issues such as static collection schedules, overflowing bins, environmental impact, and the absence of real-time monitoring. To address these problems, Higiea was conceived with a primary focus on:

- *Demand-based Operations*: Replacing rigid schedules with real-time sensor data to decide *when* and *where* to collect waste.
- *Automated Route Planning*: Reducing reliance on manual scheduling by employing an architecture that triggers new routes upon detecting `FULL` containers.
- *Scalable Informatics Infrastructure*: Ensuring that as the number of sensors and trucks grows, the system can expand without sacrificing performance or reliability.

Throughout the chapters, several key activities were undertaken to achieve these goals:

1. *Literature Review*: Investigated the state of the art in IoT-based waste management, covering sensor technologies, advanced algorithms for route optimization, and the need for robust informatics infrastructures.
2. *Proposed Solution—Higiea*: Outlined the high-level concept, main and secondary objectives, and expected outcomes. The design was driven by key non-functional requirements such as scalability, modularity, and maintainability.

3. *Planning*: Created an initial plan and methodology, later adapting it to real-world constraints, including DevOps environment setup, integration of third-party route calculators, and an iterative Kanban-style workflow.

4. *System Analysis*: Detailed a domain model for sensors, trucks, and routes. Defined functional and non-functional requirements, and described the triggers, planning strategy, and route calculation algorithms used to manage dynamic waste collection.

5. *Theoretical Background*: Covered fundamental concepts underpinning the solution, including Hexagonal (Ports & Adapters) Architecture, messaging with MQTT, and reactive programming. Each of these elements was chosen to ensure a flexible, non-blocking, and easily testable system.

6. *System Architecture*: Highlighted the end-to-end design of Higiea's backend, the dual-database persistence model (PostgreSQL + MongoDB), and the role of the MQTT broker. Emphasized horizontally scalable deployment strategies using containerization and zone partitioning.

7. *Implementation*: Showcased how the system was built using Spring Boot, focusing on hexagonal design principles, real-time message handling, and robust infrastructure adapters.

8. *Evaluation and Results*: Demonstrated the final system using a web-based simulator for sensor updates and multi-zone management, describing real-time route generation, cost analysis (cloud services plus human resources), and logging/monitoring through Azure Log Analytics.

Overall, the project successfully delivered a working proof-of-concept that meets the stated core objectives of monitoring waste container status in real time, generating routes automatically, and providing a clear, modular approach for future enhancements.

Critical Analysis: Pros and Cons

Strengths and Advantages

- *IoT-Driven Approach*: By adopting MQTT for sensor communication, Higiea handles real-time updates with minimal overhead. This ensures that even large sensor networks remain responsive.

- *Scalability*: The emphasis on horizontal scaling and zone partitioning means that a city can add more system instances, each corresponding to a different district or region, without a major redesign.

- *Hexagonal Architecture*: Separating the domain logic from infrastructure details has proven beneficial for maintainability and testability. Each port (e.g., `RouteTrigger`, `RoutePlanning`) can be replaced or upgraded independently.

- *Flexible Persistence Strategy*: Using PostgreSQL for stable relational data (trucks, sensor metadata) and MongoDB for geospatially rich route data leverages each database's strengths.
- *Production-Focused Deployment*: Containerization and integration with Azure services (Container Apps, Cosmos DB, Azure Maps) demonstrate real-world viability, reducing the gap from prototype to deployment.

Thus, while Higiea demonstrates significant strengths in scalability and modularity, it also highlights areas where future advancements could refine its functionality and real-world impact.

Limitations and Areas for Improvement

- *Route Optimization Complexity*: Current route-planning logic uses a straightforward heuristic that can handle daily operations effectively, but more sophisticated algorithms (e.g., traffic-aware routing, dynamic re-routing if a container becomes full mid-route) would enhance efficiency further.
- *Advanced Sensor Capabilities*: The system currently focuses on fill-level monitoring. Incorporating additional sensor data (e.g., temperature, presence of contaminants, or fire risk) would expand the platform's functionality and require further algorithmic complexity.
- *Edge Cases and Failover*: While horizontal scaling addresses high traffic, more resilience testing (e.g., partial network failures, broker outages) is needed to ensure consistent service under all conditions.
- *Frontend Limitations*: The simulator interface, although sufficient for demonstration, is simplistic. A more sophisticated UI/UX might be needed for large-scale operational use, especially with advanced features (e.g., driver mobile apps, real-time map overlays of traffic density).
- *Cost Optimization*: The cost analysis primarily covers baseline Azure services. Integrating advanced cost-saving techniques (e.g., auto-scaling to zero during off-peak hours, spot instances, or on-premise deployment for smaller municipalities) could be explored.

Future Work

Building on the foundations laid by Higiea, several enhancements can drive the system's evolution:

- *Predictive Analytics*: By incorporating machine learning algorithms that forecast fill-level patterns, the platform could proactively schedule pickups, further reducing the risk of overflow.

- *Edge Computing*: Offloading portions of data processing to edge devices (such as local gateways) to minimize latency, reduce cloud data transfer costs, and improve reliability when network connectivity is unstable.
- *Dynamic Rerouting*: Utilizing real-time updates from sensors and traffic data, the platform could dynamically adjust and optimize routes for collection trucks, ensuring efficient pickups while avoiding delays or unnecessary travel.
- *Advanced Frontend Tools*: A dedicated web or mobile app for truck drivers and municipal operators, featuring real-time notifications, route reassignments, and advanced analytics dashboards for higher-level decision-making.

Personal Conclusions and Reflections

From a personal standpoint, this final degree project has been an invaluable learning experience in *end-to-end software development*—from formulating a research-backed concept to implementing a production-ready prototype. Key takeaways include:

- The importance of *good architectural practices* (e.g., hexagonal design) for building maintainable systems and avoiding technical debt.
- The effectiveness of *Agile-inspired* workflows (even in a simpler Kanban style) for managing tasks and responding to evolving project needs.
- How critical *DevOps pipelines* and *CI/CD* are for ensuring continuous improvement and stable releases.
- The value of a *user-centric approach*, exemplified by the simulator, which helped me test and refine the solution's real-world usability.

Additionally, working on this project solidified my belief in the *transformative power of IoT* to address long-standing municipal challenges. Seeing how a carefully designed architecture can seamlessly integrate hardware (sensors), communications (MQTT), computation (route planning), and user interfaces (simulator) opened my eyes to the potential for more advanced applications in urban management.

Final Remarks

In conclusion, *Higiea* serves as a robust proof-of-concept for modern, data-driven waste management. By bridging IoT sensors with automated route planning, containerized cloud infrastructure, and real-time analytics, the platform demonstrates both technical feasibility and practical value. While certain limitations remain, the flexible and modular nature of the solution ensures that future work can iteratively expand its capabilities. With growing environmental concerns and urbanization pressures, an adaptable, intelligent waste management system like Higiea can contribute significantly to more sustainable and livable cities.

Appendix A

Executing the Application

This appendix provides detailed instructions on how to execute the application.

Prerequisites

Before proceeding, ensure the following are installed and properly configured:

- **Docker Desktop**: Installed and running.
- **Java 21**: Installed and added to your system's PATH.

Building the Application

1. Navigate to the root folder of the project.
2. Run the following command to build the application:

```
./gradlew build
```

Starting the Environment

1. Run the following command to start the necessary services (PostgreSQL, MongoDB, MQTT broker) and two instances of the application:

```
docker-compose up
```

2. Wait for two message:

```
HigieaApplication has started in ...
```

Accessing the Simulator

1. Navigate to the `simulator` folder.
2. Open `index.html` in your preferred browser.

Disabling Web Security (if needed)

If you encounter cross-origin resource sharing (CORS) issues, disable web security using the following steps:

- **Windows:**
  1. Locate the installation folder of `chrome.exe`.
  2. Run the following command:

```
chrome.exe –user-data-dir="C:/Chrome dev session"
            –disable-web-security
```

- **MacOS:**
  1. Open a terminal and run the following command:

```
open -a Google Chrome –args –disable-web-security –user-data-dir
```

Important Notes

- If you do not see anything on the map in the simulator, ensure web security is properly disabled as described above.
- **Route Calculation:** The application uses a mock route calculation heuristic instead of Azure Maps. To enable Azure Maps, you would need to:
  – Register with the Azure Maps service.
  – Configure the required IDs in the application environment variables

By following these steps, you will be able to successfully execute and interact with the application.

Bibliography

[1] Barcelona City Council, "Sentilo: Open Source Sensor and Actuator Platform," Barcelona City Council, 2012. [Online]. Available: https://www.sentilo.io.

[2] Bigbelly, "Smart Waste Recycling System," Bigbelly Solar, Inc. [Online]. Available: https://bigbelly.com.

[3] T. Gast, B. Atasoy, F. Duarte, and D. Rus, "An adaptive large neighbourhood search for real-time waste collection inventory routing with autonomous vessels," presented at hEART 2020, 2020.

[4] Noura, M., Atiquzzaman, M., & Gaedke, M. (2019). Interoperability in Internet of Things: Taxonomies and Open Challenges. *Mobile Networks and Applications*, *24*(3), 796–809. https://doi.org/10.1007/s11036-018-1089-9

[5] JanMM34 (October 16, 2024). Higiea. GitHub. https://github.com/JanMM34/Higiea

[6] Beck, K., et al. (2001) The Agile Manifesto. Agile Alliance. http://agilemanifesto.org/

[7] Schwaber, K. and Sutherland, J. (1991) The Scrum Guide The Definitive Guide to Scrum: The Rules of the Game. www.scrum.org

[8] Hewitt, T. (n.d.). *Semantic Versioning 2.0.0*. Semantic Versioning. Retrieved January 14, 2025, from https://semver.org/

[9] Cockburn, A. (2005). *Hexagonal Architecture*. Retrieved from https://alistaircockburn.com/hexagonal-architecture/

[10] Martin, R. C. (2000). *Design principles and design patterns* [PDF]. Retrieved from https://objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[11] Evans, E. (2003). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley. Chapter 6: The Repository, pp. 147–150.

[12] OASIS. (2019). *MQTT version 5.0: An OASIS standard*. OASIS Open. Retrieved from https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[13] Fielding, R. T., & Reschke, J. (Eds.). (2014). *Hypertext transfer protocol (HTTP/1.1): Semantics and content* (RFC 7231). Internet Engineering Task Force (IETF). Retrieved from https://www.rfc-editor.org/rfc/rfc7231

[14] Wan, Z., Hudak, P. (2000). Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 242–252). ACM.

[15] Reactive Streams. (2015). *Reactive Streams specification for asynchronous stream processing with non-blocking back pressure*. Retrieved from https://www.reactive-streams.org/

[16] Project Reactor. (n.d.). *Reactive programming library for Java*. Reactor. Retrieved from https://projectreactor.io/

[17] Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* (Doctoral dissertation). University of California, Irvine. Retrieved from https://www.ics.uci.edu/ fielding/pubs/dissertation/rest_arch_style.htm

[18] Richardson, L., & Ruby, S. (2007). *RESTful web services*. O'Reilly Media.

[19] PostgreSQL Global Development Group. (n.d.). *PostgreSQL documentation*. Retrieved from https://www.postgresql.org/docs/

[20] MongoDB, Inc. (n.d.). *MongoDB documentation*. Retrieved from https://www.mongodb.com/docs/

[21] Butler, H., Daly, M., Doyle, A., Gillies, S., Schaub, T., & Schmidt, C. (2016). *The GeoJSON format: Specification RFC 7946*. Internet Engineering Task Force (IETF). Retrieved from https://datatracker.ietf.org/doc/html/rfc7946

[22] Spring Framework. (n.d.). *Spring Framework documentation*. Spring.io. Retrieved from https://spring.io/projects/spring-framework

[23] Microsoft. (n.d.). *Azure Maps documentation*. Microsoft Azure. Retrieved from https://learn.microsoft.com/en-us/azure/azure-maps/

[24] Docker, Inc. (n.d.). *Docker documentation*. Docker. Retrieved from https://docs.docker.com/

[25] Microsoft. (n.d.). *Azure Container Registry documentation*. Microsoft Azure. Retrieved from https://learn.microsoft.com/en-us/azure/container-registry/

[26] Microsoft. (n.d.). *Azure Container Apps documentation*. Microsoft Azure. Retrieved from https://learn.microsoft.com/en-us/azure/container-apps/

[27] Microsoft. (n.d.). *Azure Database for PostgreSQL Flexible Server documentation*. Microsoft Azure. Retrieved from https://learn.microsoft.com/en-us/azure/postgresql/flexible-server/

[28] Microsoft. (n.d.). *Azure Cosmos DB API for MongoDB documentation*. Microsoft Azure. Retrieved from https://learn.microsoft.com/en-us/azure/cosmos-db/mongodb/

[29] EMQX. (n.d.). *EMQX documentation*. EMQX. Retrieved from https://www.emqx.io/docs/

[30] Dierks, T., Rescorla, E. (2008). *The Transport Layer Security (TLS) Protocol Version 1.2*. Internet Engineering Task Force (IETF). Retrieved from https://datatracker.ietf.org/doc/html/rfc5246

[31] Freier, A. O., Karlton, P., Kocher, P. C. (2011). *The Secure Sockets Layer (SSL) Protocol Version 3.0*. Internet Engineering Task Force (IETF). Retrieved from https://datatracker.ietf.org/doc/html/rfc6101

[32] Microsoft. *Azure Log Analytics Workspace Documentation*. Available at: `https://learn.microsoft.com/en-us/azure/azure-monitor/logs/log-analytics-workspace-overview`.