Waste **A**nalytics and **S**mart **T**racking **E**cosystem
`- Services Engineering and Management -`

Pedro Carneiro(73775), Inês Águia(73882) & Leandro Rito (92975)

June 14, 2025

1 Introduction

In recent years, the rapid urbanization and an increasing volume of municipal solid waste have posed significant challenges to waste collection and environmental sustainability. Traditional waste management systems often suffer from inefficiencies such as rigid collection schedules, underutilized resources, and limited real-time data for operational decision-making. To address these limitations, smart waste management solutions have emerged as a promising approach to improve efficiency, reduce operational costs, and support environmentally conscious urban development.

The `W.A.S.T.E.` (Waste Allocation and Smart Tracking Ecosystem) project proposes a modular and scalable system that leverages modern technologies—including IoT-enabled sensors, real-time data analytics, and mobile applications—to optimize the monitoring, routing, and collection of waste. The system integrates multiple components: sensor-equipped bins that measure fill levels, a backend infrastructure for data processing and route optimization, and a cross-platform Flutter application that supports different user roles, such as administrators and collectors.

This report presents the architectural design, implementation, and deployment strategies of the W.A.S.T.E. system. Emphasis is placed on the use of microservices, containerization, secure authentication, and dynamic route generation based on real-time bin data. The project also addresses interoperability between services using technologies such as Kafka, Redis, and RESTful APIs, while ensuring a seamless user experience through a responsive frontend interface.

2 System Overview

The `W.A.S.T.E.` system is built upon a `Service-Oriented Architecture (SOA)`, in which each functional unit is encapsulated as an independent and reusable service with a well-defined interface. This approach promotes modularity, scalability, and flexibility, allowing each component to evolve independently over time while ensuring seamless interoperability between services.

Each service was developed incrementally and underwent several iterations during the design and testing phases. These iterations involved performance tuning, security hardening, API restructuring, and the refinement of communication protocols (`RESTful HTTP, WebSockets and TCP`). This iterative development process allowed the system to adapt effectively to changing requirements, reduce technical debt, and align closely with real-world constraints.

Figure 1 presents a high-level architectural overview of the system. The architecture includes sensor-equipped bins, backend services for monitoring and optimization, a Kafka-based message bus, a Redis store for quick lookup operations, and a dual-layered frontend. All services are containerized and orchestrated within isolated environments, exposed securely through an API Gateway (Kong) that unifies access control, routing, and service discovery.

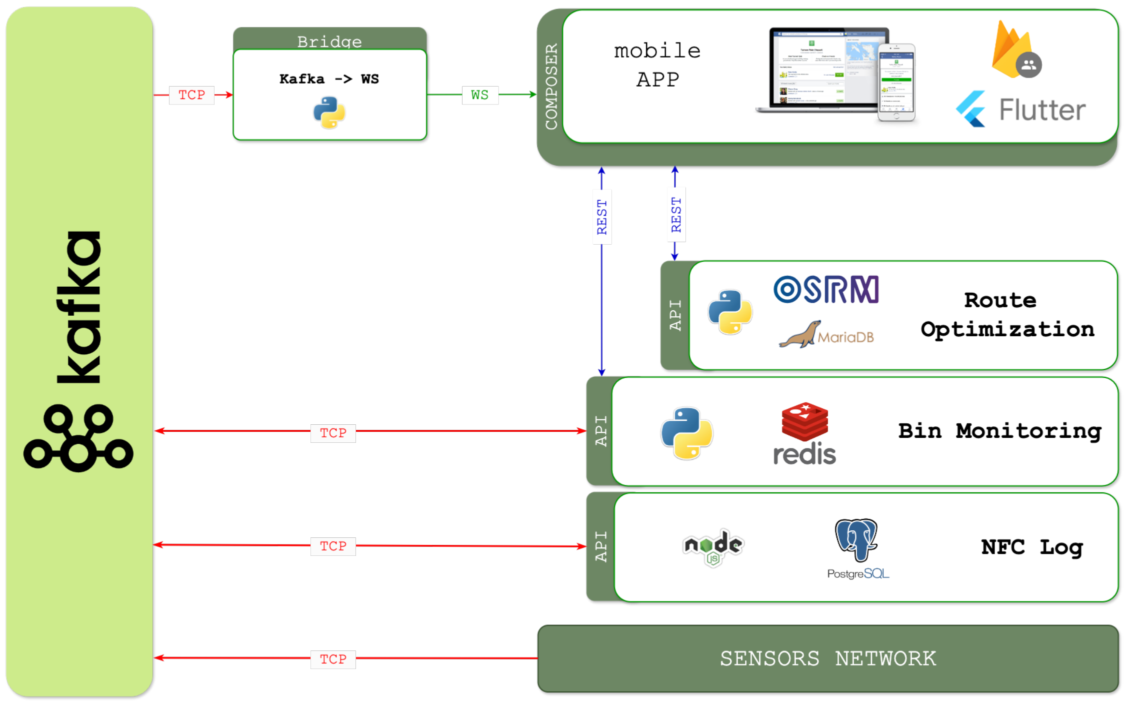

Figure 1: High-level system architecture of the W.A.S.T.E. platform. The architecture follows a Service-Oriented Architecture (SOA) model, with independent services communicating over REST and Kafka. Core components include IoT-enabled bins, backend services for bin monitoring, route optimization and NFC logging, a Redis store for fast sensor-topic mapping, and a dual-layered frontend composed of a Flutter web client and a Composer background listener. All services are containerized and exposed through an API Gateway (Kong) for unified access and routing.

The system is organized into several loosely-coupled services, each of which is described in the following subsections.

2.1 Bin Monitoring Service

The **Bin Monitoring Service** plays a dual role within the `W.A.S.T.E.` system: it is both a management layer for IoT sensor-topic associations and a message router for real-time sensor data. This service ensures that the dynamic network of smart waste bins can be monitored, updated, and maintained efficiently.

As shown in Figure 2, on the management side, the service exposes RESTful endpoints that allow for the registration, deletion, and listing of both sensors and Kafka topics. This layer provides administrators with full control over the configuration of the sensor network, enabling runtime modifications to the system without requiring redeployment.

On the routing side, the service includes a Kafka-based forwarder that listens to a shared topic where all bins periodically publish updates. Each message typically contains the sensor's serial number and the current fill level of the bin. Using a Redis-based lookup table, the service determines the appropriate topic for each sensor and republishes the message to the corresponding destination topic. This mechanism supports modular processing by downstream services and allows for scalable and topic-based message segregation.

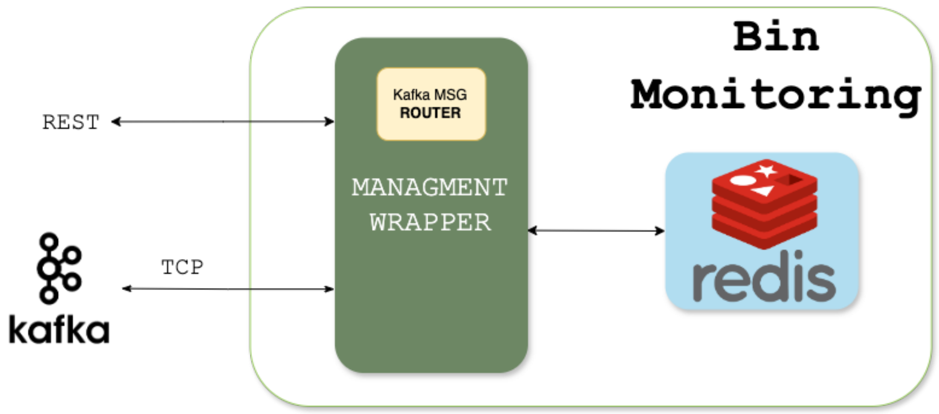

Figure 2: Internal architecture of the Bin Monitoring Service. The service acts as both a management wrapper and a message forwarder between the sensor network and downstream Kafka topics. Redis is used as a fast-access lookup table to determine the destination topic for each incoming sensor message.

Additionally, the service integrates a WebSocket bridge that pushes updates in real time to any connected client (e.g., dashboards or mobile apps).

Exposed API Endpoints Table 1 summarizes the REST API endpoints exposed by the Bin Monitoring Service under the root path `/v2/binmonitoring/`:

Table 1: REST API Endpoints exposed by the Bin Monitoring Service

Category	Endpoint	Description
Sensors	GET /sensors	Retrieves the list of all registered sensors
	POST /sensors	Registers a new sensor with its associated metadata
	DELETE /sensors	Removes an existing sensor from the system
Topics	GET /topic	Retrieves the list of topic mappings
	POST /topic	Creates or updates a topic mapping for a given sensor
	DELETE /topic	Deletes a topic mapping from the system

This modular and decoupled design allows the system to efficiently manage a growing network of bins while maintaining responsiveness and extensibility.

2.2 Route Optimization Service

The **Route Optimization Service** is responsible for generating optimized waste collection routes based on dynamic bin data, such as geolocation and fill level. It integrates a routing engine (OSRM – Open Source Routing Machine) and a MariaDB-based persistence layer to support route computation, storage, and retrieval.

The core of the service exposes a RESTful API that allows external clients to request new routes by specifying constraints such as the percentage fill threshold or a specific truck ID. The service queries the current bin state, filters the eligible bins, and submits a request to the OSRM engine for route planning. Once a route is computed, it is optionally persisted to the MariaDB database for future reference or analysis.

Figure 3 illustrates the internal architecture of the Route Optimization Service, including the communication between its core, the OSRM API, and the database layer.

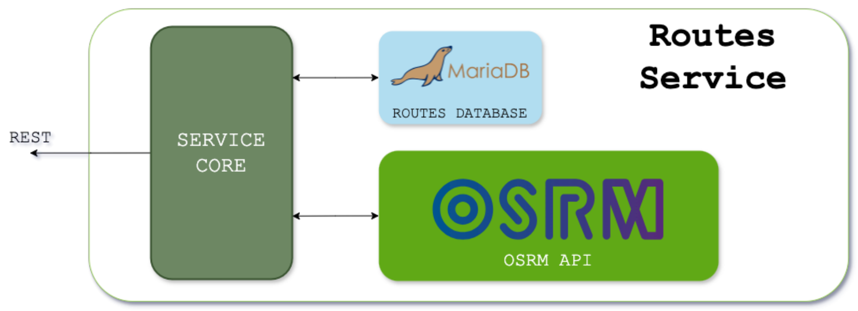

Figure 3: Internal architecture of the Route Optimization Service. The core logic interfaces with the OSRM API for route calculation and with a MariaDB instance for persistent storage of computed routes.

Exposed API Endpoints The following table summarizes the endpoints provided by the Route Optimization Service under the root path `/v2/routes`:

Table 2: REST API Endpoints exposed by the Route Optimization Service

Method	Endpoint	Description
POST	/v2/routes	Generates a new optimized route with truck ID and store it on db.
GET	/v2/routes	Retrieves the list of previously generated routes from the db.
DELETE	/v2/routes	Deletes stored routes based on input parameters or clears the history.

This modular service design enables flexible, real-time route planning that adapts to operational needs and supports intelligent waste collection.

2.3 NFC Log Service

The **NFC Log Service** is a lightweight backend module designed to collect and stream NFC-based interaction data between users and smart bins. It plays a key role in tracking user participation, which supports a usage-based incentive mechanism such as discount coupons.

The system integrates with a Flutter-based Android application that runs on users' mobile devices. When a user approaches a smart bin, the app reads the NFC tag embedded in the bin and sends a RESTful request to the NFC Log Service, including two parameters: the `nfc_tag` of the bin and the device's `IMEI`. This information is then encapsulated into a structured message and forwarded to a Kafka topic for real-time logging and downstream processing.

Figure 4 shows the architecture of the NFC Log Service, including its communication flow with the mobile app and Kafka broker.

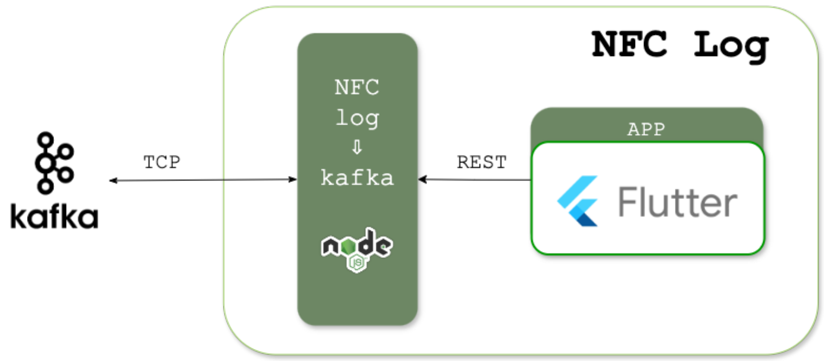

Figure 4: Architecture of the NFC Log Service. The mobile app sends NFC tag data and device identifiers to the backend via REST. The service then pushes the event to Kafka for downstream processing and analysis.

Although the service does not expose a public API for querying logs, its integration with Kafka enables real-time event propagation to other services, such as usage analytics or user dashboards. This enables the system to track individual usage and associate actions with specific users, paving the way for a coupon or credit-based reward system.

2.4 Frontend Architecture

This architecture follows a strict separation of concerns between the user interface and backend logic. The interface is built using a **Flutter Mobile App**, offering a responsive and cross-platform experience. All data persistence and WebSocket integration are handled externally by backend services.

User authentication is managed via **Firebase Authentication**, supporting both Google and GitHub login providers. These mechanisms are implemented using dedicated Flutter packages, ensuring secure and seamless integration within the app.

As illustrated in Figure 5, the Flutter application does not interact directly with Kafka. Instead, a dedicated Python-based **bridge service** was developed to convert Kafka messages into WebSocket-compatible events. This bridge listens to specific Kafka topics and forwards relevant updates to connected clients over standard WebSocket channels.

All persistent interactions (such as user registration, bin management, or usage tracking) are performed via RESTful APIs exposed by the **Composer** backend. This backend communicates with a MariaDB database and exposes a structured set of endpoints under the root path `/v2/api/`.

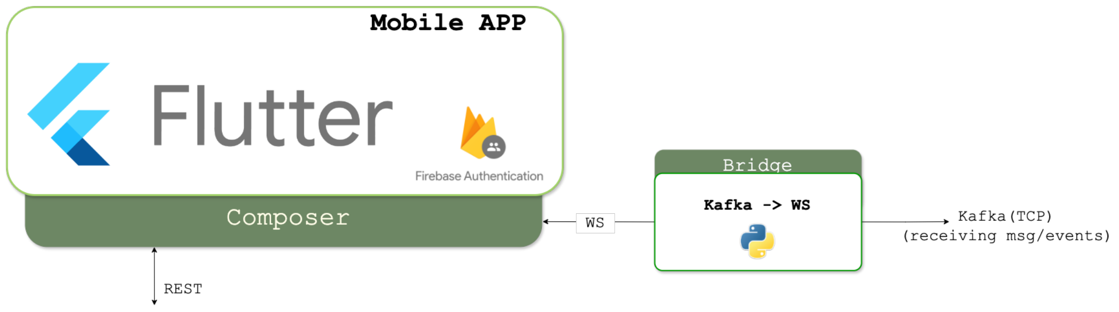

Figure 5: Updated frontend architecture. The Flutter mobile app authenticates users via Firebase (Google and GitHub) and interacts with the Composer backend via REST. Real-time updates are delivered through a Kafka-to-WebSocket Python bridge.

Exposed API Endpoints (Composer Service) The Composer backend exposes the following REST endpoints under `/v2/api/`, grouped by functionality:

Table 3: REST API Endpoints exposed by the Composer Backend

Category	Endpoint	Description
Bins	GET /bins	Retrieves all registered bins
	POST /bins	Registers a new bin with metadata
	PUT /bins/{bins_id}	Updates the properties of a specific bin
	DELETE /bins/{bins_id}	Removes a bin from the system
	PUT /bins/fill/{sensor_serial}	Updates the fill level of a bin
Users	GET /users	Lists all registered users
	POST /users	Creates a new user
	GET /users/{uid}	Retrieves information about a specific user
	DELETE /users/{uid}	Deletes a user from the system
	PUT /users/{uid}/imei	Updates the IMEI associated with a user
	POST /users/{uid}/increment_usage	Increments the usage counter of the user
	GET /users/{uid}/usage_count	Returns the number of bin uses per user
	GET /users/exists/{uid}	Checks whether a user exists in the system
	GET /users/by_imei/{imei}	Retrieves user information by IMEI
	POST /users/{uid}/reset_usage	Resets the usage counter for a user

3 Deployment

All services in the `W.A.S.T.E.` platform are deployed within the **Kubernetes cluster maintained by the University of Aveiro (UA)**, under a dedicated namespace named `waste-app`. Each core module—ranging from data ingestion to frontend delivery—is containerized and deployed as a separate pod, with appropriate service discovery, volume claims, and ingress routing configured for stable operation.

Docker images for each service were pre-built and pushed to the university's private container registry. These images were then pulled by the cluster and orchestrated within the defined namespace using YAML manifests.

To overcome the limitation that external clients cannot access Kafka directly from outside the cluster, a dedicated RESTful abstraction layer was developed and deployed in a pod named `posting`. This microservice allows external clients to post data via HTTP, which is then relayed to Kafka within the cluster boundary.

Moreover, all MariaDB-based services rely on **Persistent Volume Claims (PVCs)** to guarantee data durability and consistent storage across pod restarts and migrations.

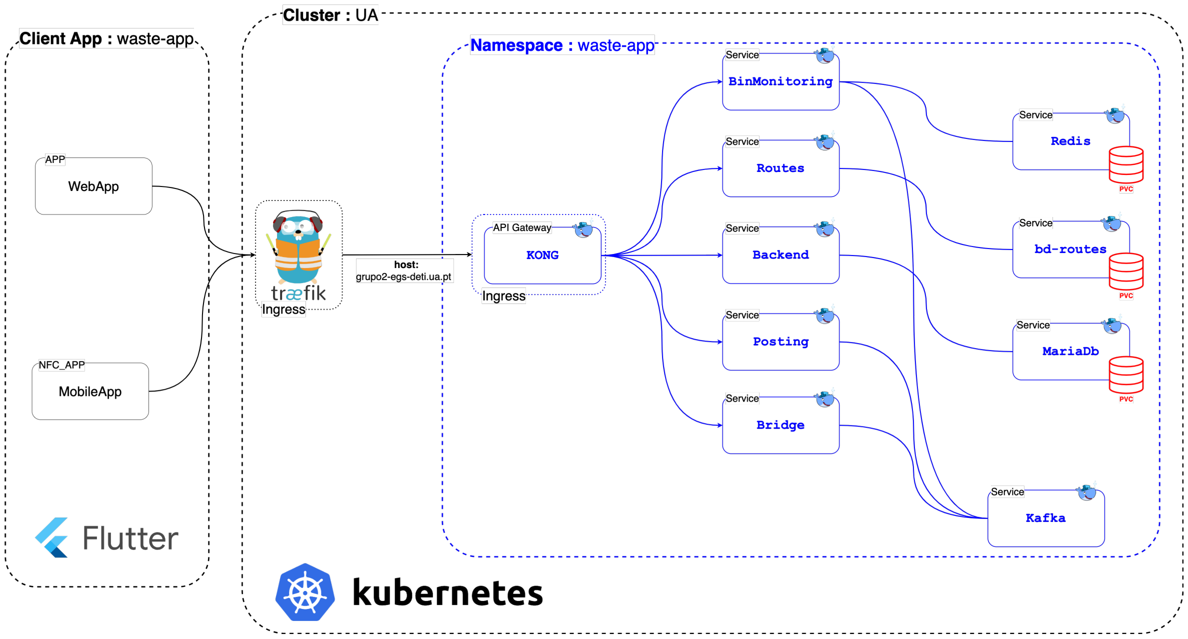

Figure 6: Deployment architecture of the W.A.S.T.E. platform in the Kubernetes cluster of the University of Aveiro. Each component is deployed in the `waste-app` namespace with dedicated services and ingress points.

Pods and Their Roles

- **posting**: Exposes a REST API that receives external messages and internally publishes them to Kafka topics.
- **routesAPI**: Handles route optimization and retrieval using OSRM and MariaDB.
- **db-routes**: MariaDB instance used by the route optimization service.
- **binMonitoring**: Ingests bin sensor data, manages sensor-topic mappings, and forwards messages to Kafka.
- **kafka**: Message broker used for decoupling and streaming between services.
- **redis**: In-memory store.
- **kong**: API Gateway managing access to all internal APIs.
- **backend**: Composer service exposing user and bin management endpoints.
- **bridge**: WebSocket bridge that receives Kafka messages and pushes them to clients.
- **mariadb**: Central database for the Composer service, used for user and bin persistence.

Deployment Overview The table below summarizes the main components of the deployment, along with their configuration elements:

Table 4: Summary of Kubernetes components required to deploy each service in the `waste-app` namespace. A green checkmark (✓) indicates that the corresponding Kubernetes object was created and is essential for the proper operation of the service. Each column represents a specific resource type—such as `Deployment`, `Service`, `ConfigMap`, or `Ingress`—and each row corresponds to a deployed pod. Empty cells indicate that the resource was not needed for that service.

Pod	**Deployment**	**Service**	**ConfigMap**	**Secret**	**Ingress**	**Volumes**	**Working**
posting	✓	✓					✓
routesAPI	✓	✓	✓	✓	✓		✓
db-routes	✓	✓				`PVC`	✓
binMonitoring	✓	✓	✓	✓	✓		✓
kafka	✓	✓					✓
redis	✓	✓				`PVC`	✓
kong	✓	✓	✓		✓		✓
backend	✓	✓		✓	✓		✓
bridge	✓	✓		✓			✓
mariadb	✓	✓	✓			`PVC`	✓

Although individual `Ingress` resources were defined for several RESTful services during the initial development phase, these were primarily used for debugging and direct testing. In the final deployment, all external HTTP traffic is routed through a single entry point managed by `Kong`, which acts as an API Gateway. As such, only the `Kong` ingress remains active and functional, abstracting access to the underlying services and centralizing authentication, routing, and traffic control within the `waste-app` namespace.

4 Tests and Validation

To validate the correct functioning of the `waste-app` system, a series of targeted tests were conducted, covering each of the critical components and their interactions.

Sensor Simulation A Python-based simulation script was developed to emulate the behavior of bin sensors. This simulator generates and sends realistic data regarding bin fill levels, mimicking the communication flow of physical devices. The simulation confirmed that the system is capable of receiving, processing, and reacting to sensor updates in real time via the Kafka pipeline, which in turn feeds data to the WebSocket bridge and updates the frontend interface accordingly. This validated the end-to-end data flow and confirmed the responsiveness and correctness of the monitoring infrastructure.

NFC Interaction To evaluate the NFC subsystem, tests were performed using mobile devices capable of NFC communication. The system successfully captured both the NFC tag ID and the device's IMEI, which were correctly sent to the backend through the configured RESTful API. This confirms that the infrastructure is ready to be integrated with a physical prototype where user authentication or interaction is performed via NFC tags, providing a foundation for future real-world deployments.

Service Functionality All backend services were individually tested via their REST APIs, ensuring full coverage of the intended features:

- Successful creation and retrieval of user accounts with role-based access control.
- Registration and management of bins, including associating them with sensors and updating their configuration dynamically.
- Generation and deletion of optimized waste collection routes using the OSRM-based routing service.
- Verification of data persistence and integrity within the MariaDB databases across multiple services.

- Secure communication via services protected by secrets and configuration maps, with successful integration through the Kong API Gateway.

Overall, the tests demonstrate that the system operates as intended under simulated conditions, and that the integration between components is robust and modular, ensuring scalability and maintainability for future development and deployment phases.

5 Conclusion

This project presents a modular and scalable system for smart waste management, combining real-time monitoring, route optimization, and user interaction through NFC. The architecture is based on a SOA approach deployed on a Kubernetes cluster, ensuring high availability, maintainability, and flexibility. Through the integration of RESTful APIs, WebSocket communication, and containerized services such as Kafka, MariaDB, and Kong Gateway, the system supports dynamic bin tracking and user engagement in a city-scale deployment scenario.

The implementation was validated through comprehensive testing, including simulated sensor inputs and NFC interactions, confirming the overall robustness and correctness of the solution. Moreover, the frontend, developed in Flutter, offers a cross-platform experience and ensures seamless communication with backend services.

This work demonstrates the feasibility of a distributed waste tracking system supported by cloud-native technologies and outlines a strong foundation for future expansion.

5.1 Future Work

During the project presentation and internal review, several directions for future development were identified:

- **Proximity-based Bin List:** Currently, the bin list presented to the user is not ordered by spatial relevance. A future improvement would involve ordering bins based on their geographic distance to the user's current location, enhancing usability and efficiency in route planning.
- **Reward System Reassessment:** The current coupon and reward system is functional but simplistic. Future iterations should explore more engaging gamification strategies, possibly incorporating dynamic incentives based on user participation levels or environmental impact.
- **Real-Time Truck Tracking:** A promising feature would be the integration of GPS tracking for garbage trucks. This would allow real-time visualization of waste collection processes, enabling improved transparency and logistical coordination.
- **Unified Application Interface:** At present, the system is divided between an NFC-capable Android application and a Flutter-based web frontend. A key objective for future development is to unify both components into a single mobile application that supports all functionalities, improving usability and lowering the barrier to adoption.
- **User Feedback Mechanism:** Enabling users to submit feedback regarding their experience, issues encountered, or suggestions for improvement would provide valuable insights to guide future iterations of the platform and strengthen community engagement.

These proposed enhancements aim to increase system utility, user satisfaction, and operational efficiency, guiding the platform towards a mature and deployable smart city solution.

6 Auto-Evaluation

Last but not least, our self-assessment for the work presented based on our commitment and the quality of the outcome is a grade of 17.

The link for the project's repository can be found here: `https://github.com/PedroMiguelTorresCarneiro/W.A.S.T.E.`