Report: Roadsense

Edge Computing in the IoT

Luca Di Bello, Paolo Deidda, Nawfal Abdul Malick, Georg Meyer

Semester-project

Due: 20 December 2024

Contents

1 System Design **3**
1.1 Overview . . . 4
1.2 Sensor Nodes . . . 4
1.3 System Architecture . . . 6
1.3.1 IoT Data Pipeline . . . 6
1.3.2 Client-Server interaction . . . 8

2 System Implementation **9**
2.1 Prototype Car . . . 10
2.2 Prototype Embedded Firmware . . . 10
2.2.1 Mainfile (roadsense-embedded.ino) . . . 11
2.2.2 roadqualifier.h . . . 12
2.2.3 RabbitMQClient.h . . . 13
2.3 Prototype data processing pipeline . . . 14
2.4 Prototype Frontend . . . 14

3 Results **15**
3.1 Demonstrations . . . 15

List of Figures

1.1 RT-UML of a Sensor Node 4
1.2 IoT Data Pipeline 7
1.3 Client-Server Interaction 8

2.1 RoadSense Prototype RC Top View 10
2.2 Prototype Data Processing Pipeline Architecture 14

3.1 Exemplary Datavisualization with Roadsense Web Application 15

Chapter 1

System Design

The **RoadSense** project seeks to develop an IoT-powered system for detecting and mapping road anomalies, such as potholes and uneven surfaces. By equipping multiple vehicles with sensor nodes, the system will gather and analyze road vibration data to generate an interactive, detailed heatmap of road conditions. This data will be instrumental in optimizing road maintenance, enhancing driver safety, and providing real-time hazard alerts.

The project aims to deliver a comprehensive solution for road condition monitoring by addressing key objectives across data collection, processing, visualization, and alerting. Specifically, the system will focus on:

1. Designing a cost-effective IoT-based solution for detecting and mapping road anomalies.
2. Measuring road bumpiness and issuing real-time alerts for hazardous conditions.
3. Providing a user-friendly interface for stakeholders to visualize road conditions and manage alerts effectively.
4. Enhancing road condition accuracy through data collection from multiple vehicles.

In the following sections, we will delve into the system design of **RoadSense** project and explore the design choices, components, and technologies employed to achieve these objectives.

1.1 Overview

The **RoadSense** system consists of the following components:

- **Sensor Nodes**: IoT devices installed in vehicles, responsible for collecting inertial data using an Inertial Measurement Unit (IMU) sensor and location data via a GPS module. These nodes pre-process data to compute a qualifier for localized road states, reducing the volume of data sent to the central infrastructure.
- **Data Aggregation and Processing System**: A centralized backend platform responsible for receiving, aggregating, and analyzing data from multiple sensor nodes. This system generates detailed road quality insights and produces interactive heatmaps for visualization. It also includes mechanisms for detecting anomalies and triggering alerts.
- **Control Logic**: Defines the operational behavior of the IoT devices, including protocols for data collection, processing, and communication with the central system.
- **User Interface**: An interactive web application that enables stakeholders to visualize road conditions, explore heatmaps, and manage alerts effectively.

1.2 Sensor Nodes

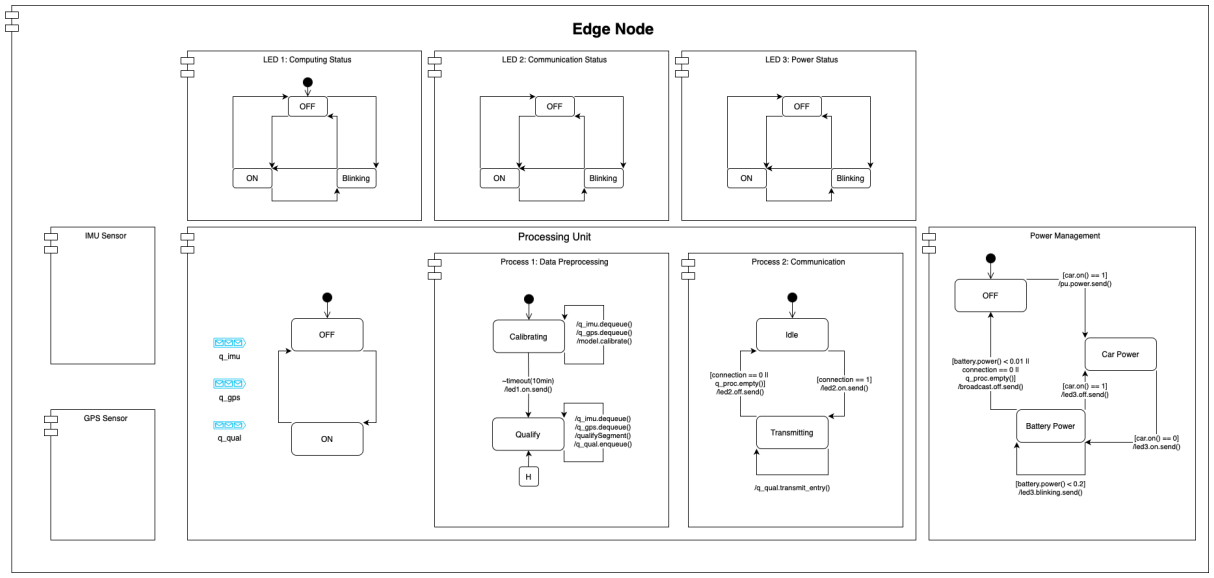


Figure 1.1: RT-UML of a Sensor Node

Main Design Aspects

1. **Cost Restriction per node**: 100 CHF
   Given the large number of vehicles that will host sensor nodes, the cost per node must remain as low as possible. To achieve this, each vehicle will have a single sensor node/package installed to minimize installation and part costs. The qualification

model is designed to be computationally efficient to minimize the cost of the microcontroller. Wifi connectability is chosen to minimize the cost of the communication module.

2. **Quantification of Road State**:
   The sensor node will be ideally positioned centrally in the vehicle, above one of the axles, and securely mounted to the chassis to reduce measurement errors. The road state will be quantified on a scale from 0 (very good) to 244 (very poor), with 255 reserved for hazardous conditions.

3. **Qualification Model**:

$$\text{RoadQuality}_i = \lfloor \frac{max(\Delta a_{z,t}) - \Delta a_{z,\min}}{\Delta a_{z,\max} - \Delta a_{z,\min}} \rfloor \cdot 255 \quad \text{where} \quad t \in \text{RoadSegment}_i \quad (1.1)$$

   This model is based on the assumption that the maximum acceleration in z-axis is proportional to the road quality. By calculating the acceleration difference between the current and the previous time step, the model is unaffected by the vehicle's orientation and acceleration. $\Delta a_{z,\min}$ and $\Delta a_{z,\max}$ are determined during the calibration phase and symbolize the minimum and maximum acceleration difference occurring during possible driving conditions. This makes the model adaptable to different vehicles and driving conditions.

   Future iterations will adapt a simulation based approach described in the following: A simple linear Mass-Spring-Damper Model is chosen to model the cars factor on the transduced shocks. (While keeping computational effort low.) A first calibration phase coupled to a initial parameter set aims to fit Mass-Spring-Damper Model parameters. Measured data will be fit to quantified values during calibration phase. Further physical quatities other than z-axis acceleration have to be considered to decouple driving induced accelerations from the road state.

4. **High Polling Rate for IMU Measurements**:
   Road-induced shocks are brief and their period and amplitude are proportional to vehicle speed. The IMU's polling rate will be configured to ensure reliable readings for typical driving speeds. Currently the acceleration is measured every 3ms, acchieving a road resolution of 2.5cm at 30 km/h which is the assumed avrage speed in urban areas.

5. **Sensing of physical quantities**: The system will measure multiple physical quantities to ensure accurate road state assessments:

   (a) **Acceleration in z-Axis** to determine road state and potholes.

   (b) **Acceleration in x,y-Axis and rotational acceleration** to minimize errors induced from driving scenarios. (Possible part of future work)

   (c) **Driving Velocity** to approximate relative distance through integration needed for velocity indipendent Segmentation of QualityMeasures.
   (Future Work: to couple shock amplitudes to velocity through Spring-Damper Model).

(d) **Geographical Position** to reference qualification to current position.

6. **Data Transmission at Established Gatepoints**:

   (a) **Data Format**: Each data package will include the following information encoded as a JSON object:

   ```
   (Node ID (2 Bytes)) |
    Position (2 x 8 Bytes (Double-Precision Float)) |
       Road Quality (1 Byte) | Unix Timestamp (4 Bytes)
   ```

   For example, the following snippet represents a valid data sample in JSON format:

   ```
   {
   "lat": 46.19313,
   "lon": 6.80421,
   "timestamp": 1734478933,
   "bumpiness": 50,
   "device_id": "USI-Car-1""
   }
   ```

   (b) **Local Preprocessing**: The node will preprocess and store position-quality tuples locally.

   (c) **Gatepoint Connectivity**: The node will automatically establish a connection at predefined gatepoints to transmit new data.

   (d) **Data Protocol**: Data packages will be transmitted in MQTT format to a RabbitMQ server.

1.3 System Architecture

The **RoadSense** consists of multiple IoT devices installed in vehicles, communicating with a central server designed to be highly scalable to handle data from thousands of devices. In this section we will describe all meaningful components of the system, focusing on the IoT data pipeline and the client-server architecture.

1.3.1 IoT Data Pipeline

The data pipeline has been designed with scalability in mind, allowing for efficient data collection, processing, and data analysis from multiple (potentially thousands) concurrent IoT devices. The following diagram illustrates the pipeline steps, from data ingestion to the storage of processed data.

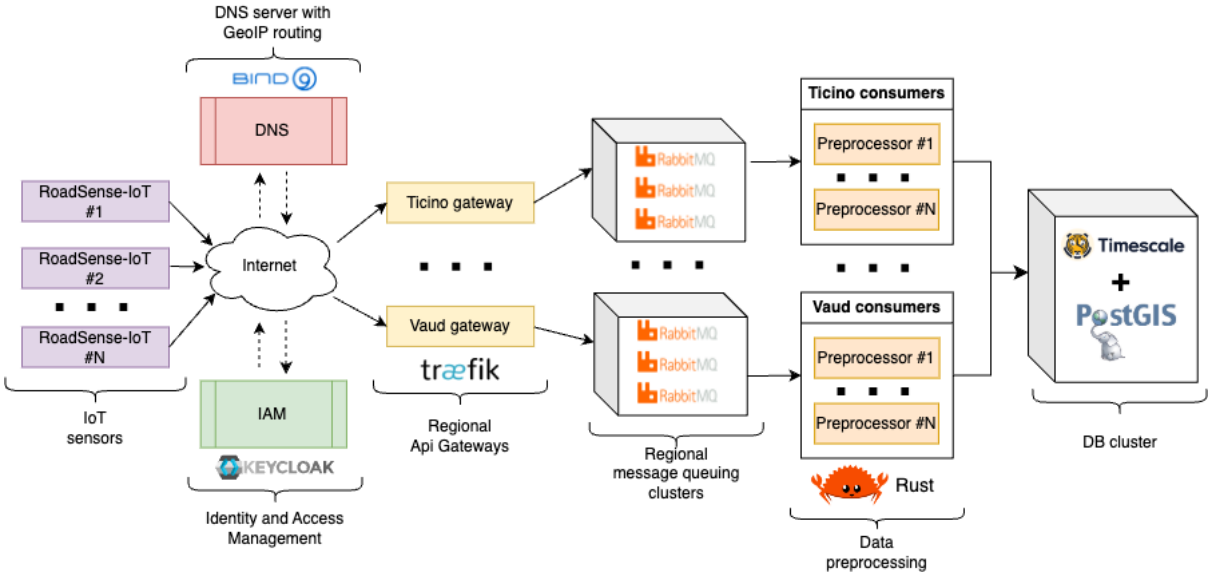


Figure 1.2: IoT Data Pipeline

The pipeline consists of the following components:

1. **Data ingestion**: IoT devices collect vibration, GPS, and other relevant data points. When the vehicle reaches an access point, the data will be transmitted to the server. Each device will be able to connect to different Wi-Fi networks, allowing for data transmission in different locations. This may include public Wi-Fi networks, cellular data, or a dedicated network infrastructure.

2. **Authentication and security**: In the original idea of the project each device is authenticated before data transmission to ensure data integrity and prevent unauthorized access. For this purpose, was decided to use Keycloak for identity and access management. Unfortunately, this feature was not implemented in the presented prototype in order to focus on the core functionality of the system.

3. **Geographical distribution**: The server leverages DNS-based load balancing to distribute incoming data across regional gateways for efficient processing. We have chosen to use BIND9 for DNS-based load balancing along with GeoIP for geolocation. Each regional gateway will be responsible for routing the user requests to the regional message queuing system. For this purpose, we will use Traefik as the reverse proxy. Unfortunately, also this feature was not implemented in the prototype as it would introduce additional complexity to the system. However, this feature is essential for the scalability of the system as it allows for efficient data processing across multiple regions.

4. **Message queues**: Each gateway node processes incoming data and forwards it to a regional queuing system to allow for parallel processing. After evaluating multiple options, we decided to use RabbitMQ as the message queue system. To ensure high availability, we will deploy RabbitMQ in a cluster configuration (refer to the RabbitMQ Clustering Guide). For the prototype we avoided the creation of a RabbitMQ cluster, and was used a single instance of RabbitMQ. This feature is still of interest for the scalability of the system.

5. **Data preprocessing**: Each region has a set of preprocessing microservices that consume incoming data from the regional message queuing system, perform data validation, and run initial data processing tasks. These microservices are deployed using containerization technology like Docker and in a future production environment, managed by Kubernetes. During the initial specification of the project was decided to use Go as the primary language for these microservices. However, in the prototype was used Rust as the primary language for the microservices. This choice was made to explore the performance and safety features of Rust. The microservices were designed to be lightweight, efficient and fail-safe.

6. **Data storage**: Processed data is stored in a scalable database system that can handle high volumes of data. Since we are dealing with both date-time and geospatial data, we chose to use TimescaleDB as the database system with the PostGIS extension to support geospatial queries. PostGIS was used to store and query collected samples

1.3.2 Client-Server interaction

The **RoadSense** system employs a client-server architecture designed to efficiently deliver real-time road condition data. The client is a web-based application that visualizes road condition samples on an interactive map. These samples are fetched from a custom API microservice, which only returns data points within the map's current bounding box, leveraging **PostGIS** for spatial queries to optimize performance and minimize data transfer. This ensures scalability and efficient handling of large datasets, as the system can support millions of samples without overwhelming either the client or the server. The following diagram illustrates the client-server interaction:

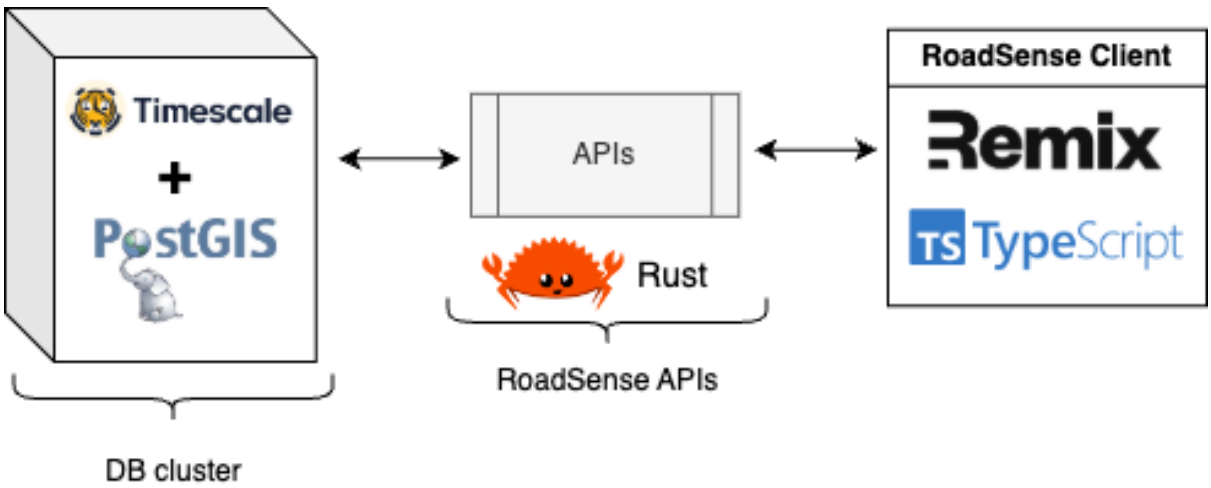


Figure 1.3: Client-Server Interaction

The backend is implemented as a custom API server built with **Rust** using the **Actix** web framework and Diesel for database interaction. It provides a read-only interface, returning road condition samples in `JSON` format based on client requests. Data insertion is handled by separate consumers processing messages from **RabbitMQ**. The architecture enables efficient data processing, retrieval, and real-time visualization while maintaining scalability and performance.

Chapter 2

System Implementation

This chapter details the implementation of the **RoadSense** system, focusing on the practical realization of its architecture and components. The system integrates various technologies to ensure reliable data collection, processing, and visualization for monitoring road conditions.

The implementation covers three main areas: the IoT sensor nodes deployed in vehicles for data acquisition, the backend infrastructure responsible for data aggregation and analysis, and the client-side application used for data visualization and user interaction. Each component is designed to optimize performance, scalability, and usability.

The IoT sensor nodes preprocess data locally to reduce transmission overhead while ensuring accurate representation of road states. The backend, built using a microservice-based approach, processes incoming data, stores it efficiently, and provides APIs for real-time access to relevant datasets. The client application uses these APIs to present road condition data interactively on a map, supporting features such as filtering, heatmaps, and severity-based color coding.

This chapter provides detailed insights into the implementation of these components, explaining the choices of technologies and methodologies employed to achieve the desired functionality and performance of the system.

2.1 Prototype Car

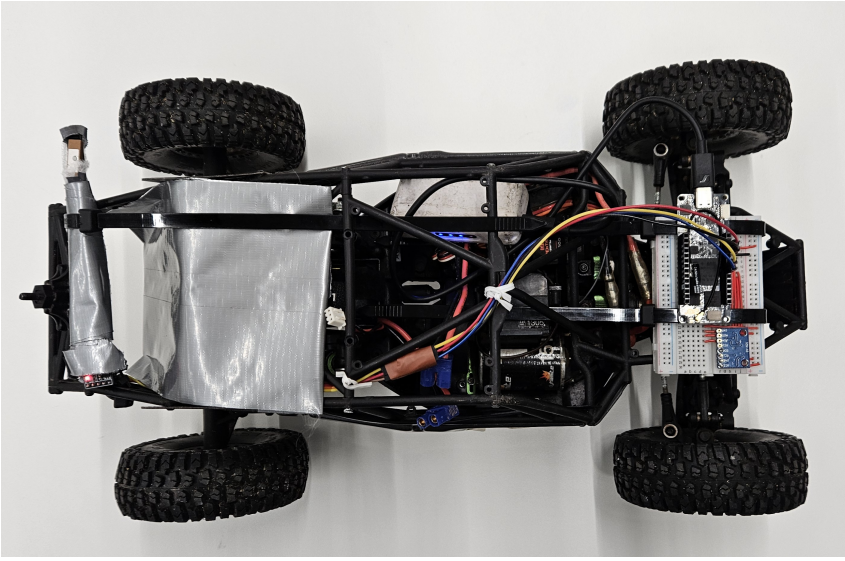


Figure 2.1: RoadSense Prototype RC Top View

Hardware Components

1. **Microcontroller**:
   Arduino Portenta H7 with built-in Wi-Fi capability.

2. **IMU Sensor**:
   GY-521 with MPU6050 6DOF (3-Axis Gyro and 3-Axis Accelerometer).

3. **GPS Module**:
   DFRobot GPS + BDS BeiDou with output of position and speed.

4. **EMF Shielding**:
   DIY using aluminum foil to shield the GPS module from electromagnetic interference.

Wiring

2.2 Prototype Embedded Firmware

In this section, we provide an overview of the core components and implementation details of the prototype embedded firmware developed for the sensor node. The firmware orchestrates sensor data acquisition, road quality analysis, and reliable data transmission to an external system. The following subsections summarize the primary files and their responsibilities:

- **Mainfile (roadsense-embedded.ino):** Initializes the system, sets up multithreaded operations, and manages the data flow between sensor acquisition and network transmission.

	Sensor	Portenta H7	Description
GY-521	VCC	3.3V	Power supply (3.3V)
	GND	GND	Ground
	SDA	SDA (Pin 11)	I2C Data line (SDA)
	SCL	SCL (Pin 12)	I2C Clock line (SCL)
DFRobot GPS	VCC	3.3V	Power supply (3.3V)
	GND	GND	Ground
	TX	RX (Pin 13)	Serial data transmit line (TX from GPS to RX on Portenta H7)
	RX	TX (Pin 14)	Serial data receive line (RX from GPS to TX on Portenta H7)

Table 2.1: Pin Connections for Sensors with Arduino Portenta H7

- **roadqualifier.h:** Contains the logic for measuring, calibrating, and quantifying road segment quality using integrated sensors, along with persistent calibration data handling.
- **RabbitMQClient.h:** Handles WiFi connectivity and MQTT-based communication, enabling the sending of computed road quality metrics to a RabbitMQ server.

By clearly defining these components, the firmware maintains a modular structure, simplifying development, testing, and future enhancements.

2.2.1 Mainfile (roadsense-embedded.ino)

The main Arduino `roadsense-embedded.ino` file serves as the central entry point for the embedded firmware running on the sensor node. Its primary tasks involve initializing system components, orchestrating two concurrent threads for road data acquisition and transmission, and managing communication buffers.

- **Initialization and Setup:** At startup, the main file initializes serial communication for debugging. It then sets up the `RoadQualifier` instance, which prepares sensor input (e.g., IMU and GPS readings) for analyzing road quality. If initialization fails, the system reports this via serial output.
- **Multithreading using Mbed OS:** Leveraging Mbed OS RTOS features, the firmware runs two threads concurrently:
  1. *Road Segmentation Thread*: Periodically calls `roadQualifier.qualifySegment()` to compute the quality of a road segment. Upon success, it stores the resulting `SegmentQuality` record into a thread-safe circular buffer.
  2. *Data Transmission Thread*: Establishes and maintains a WiFi connection, then continuously reads from the circular buffer to transmit data using a `RabbitMQClient`. If no data is available, it waits until new records arrive.
- **Circular Buffer for Data Storage:**
  A custom circular buffer, protected by a mutex, ensures safe concurrent access from both threads. If the buffer is full, the oldest entry is overwritten, preventing blocking conditions and ensuring efficient memory usage.

- **Data Transmission via RabbitMQ:**
  Once connected to WiFi, the data transmission thread publishes buffered `SegmentQuality` records to an external system through the `rabbitMQClient`. This design decouples data acquisition from network-related issues, allowing both to operate independently.
- **Watchdog and Timing:**
  Although not fully explored in the provided snippet, the code includes a watchdog timer and uses `ThisThread::sleep_for()` to handle timing and maintain system responsiveness.
- **Main Loop:**
  The `loop()` function remains empty, as the system relies on RTOS threads for ongoing tasks. All main logic thus resides in separate threads defined in the setup phase.

2.2.2 roadqualifier.h

The `roadqualifier.h` file encapsulates the logic and data structures required to process road quality measurements from connected sensors, manage calibration and data persistence, and ensure system readiness. This file defines the `RoadQualifier` class, which serves as the core of the road quality analysis functionality.

- **Sensor Abstraction and Dummy Modes:** The code supports both actual hardware operation and dummy sensor modes for testing without physical IMU or GPS devices. Conditional compilation flags (e.g., `DUMMY_MPU` and `DUMMY_GPS`) select between real and simulated sensor inputs. This approach allows for development and debugging of other modules without actual available sensors.
- **Road Segment Qualification:** The `RoadQualifier` class provides a `qualifySegment()` method to measure a predefined road segment's quality. It uses acceleration data (from the MPU6050 or dummy equivalent) and position/speed data (from a GPS module or dummy object) to compute a `SegmentQuality` metric. If a valid segment is detected, it returns a quantized quality value mapped into a byte range.
- **Calibration Handling and Flash Memory:** The file includes routines for:
  - *Calibration*: Acquiring accelerometer data over a specified timeframe to determine minimum and maximum values, ensuring that subsequent measurements are interpreted correctly.
  - *Persistent Storage*: Using Mbed's `FlashIAPBlockDevice` and related helpers (`FlashIAPLimits.h`) to store and retrieve calibration parameters (e.g., minimum and maximum acceleration differences) in non-volatile flash memory.
  - *Deletion of Calibration Data*: Providing a function `deleteCalibrationFromFlash()` to erase previously stored calibration information, enabling reset or re-calibration scenarios.
- **Quantification and Mapping:** A dedicated `quantifyToByte()` function maps computed acceleration differences into a 0–255 byte range based on the caputred calibration data. This allows for easy interpretation, efficient storage and transmission of road quality metrics.

- **Initial Setup and Readiness Checks:** The `begin()` method initializes sensors, loads or creates calibration data, and ensures a stable GPS fix before considering the system ready. The `isReady()` method provides a quick way to confirm that the `RoadQualifier` is fully operational.
- **GPS and IMU Integration:** Functions such as `waitForValidLocation()` and `waitForValidSpeed()` ensure that the system obtains reliable, fresh data from the GPS before proceeding. The IMU (or dummy MPU) data is read at each iteration, feeding the computation that identifies peak acceleration differences along the measured road segment.

2.2.3 RabbitMQClient.h

The `RabbitMQClient.h` file manages the communication between the sensor node and an external RabbitMQ server over MQTT. It encapsulates WiFi connectivity handling, MQTT client operations, and the formatting and publishing of road segment data into a consistent interface.

- **WiFi Connectivity Management:** The class attempts to connect to one of several predefined WiFi networks. It continually checks WiFi status and provides a method `isConnectedWiFi()` to confirm a successful connection. By iterating through a list of credentials, the code increases the likelihood of establishing a network connection in various deployment environments.
- **MQTT Integration for RabbitMQ:** The `RabbitMQClient` uses the `PubSubClient` library to communicate over the MQTT protocol. It sets up the MQTT server (RabbitMQ host, port, user, and password) and ensures a persistent connection. The `connect()` method and the internal `ensureConnected()` helper function handle reconnection logic and error reporting.
- **Error Handling:** In case of connection failures or publishing errors, the class stores the MQTT state code, accessible via `getErrorCode()`. This mechanism aids in debugging and understanding the cause of communication issues.
- **Publishing Data and Callbacks:** To send road segment quality data, the class provides:
  - *`publishSegmentQuality()`*: Converts a `SegmentQuality` struct into a JSON-formatted message and publishes it to a designated MQTT topic.
  - *`sendDataCallback()`*: A method suitable for periodic or callback-driven operations, connecting to the RabbitMQ server (if not connected) and publishing freshly acquired segment data.
- **Integration with the Firmware:** By abstracting away the details of WiFi and MQTT connections, `RabbitMQClient` allows other parts of the firmware—such as the road qualifier threads—to focus solely on data acquisition and retrieval. The communication logic remains modular, enabling future changes to the network stack or message format without altering the core road quality logic.

2.3 Prototype data processing pipeline

The prototype implementation of the data processing pipeline is a simplified version of the architecture described in subsection 1.3.1. Certain components such as **Keycloak** for authentication, geographical-based routing, and running services in a cluster have been omitted. These decisions were made to streamline development and focus on the core functionality of the system, deferring concerns like scalability and advanced security to future iterations.

The core concept remains consistent with the original design: a **queuing server** is placed behind a **reverse proxy** (using **Traefik**) to provide enhanced security and additional features such as load balancing and request routing. A **consumer** service then pulls data from the queue, performs preprocessing, and stores the processed data in a database. This approach enables modularity and ensures the data is prepared for subsequent analysis and visualization.

While the current implementation lacks certain advanced features, it retains the essential components to validate the core functionality. This includes the ability to handle incoming IoT data, preprocess it, and store it in a format optimized for the system's use cases. The pipeline serves as a foundation for future iterations, where scalability, geographical routing, and authentication mechanisms can be incorporated.

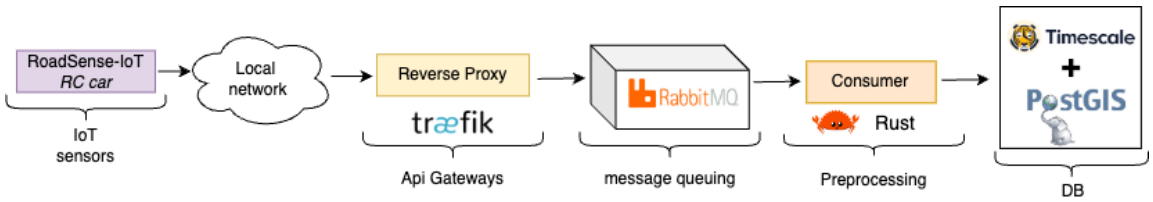


Figure 2.2: Prototype Data Processing Pipeline Architecture

This simplified implementation allows for faster prototyping and development while maintaining a clear path for future enhancements to address scalability and security concerns.

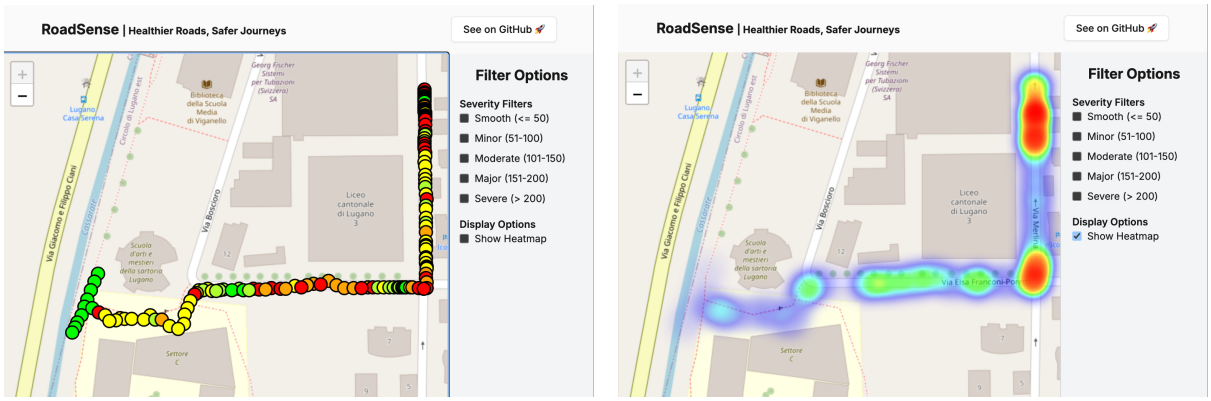
2.4 Prototype Frontend

<todo>

Chapter 3

Results

3.1 Demonstrations



(a) Point Quality Data (b) Data Distribution Heatmap

Figure 3.1: Exemplary Datavisualization with Roadsense Web Application