IoCarT - vehicle monitoring for driving school

Miggiano Davide 4840761
Morando Andrea 4604844

Introduction

In the context of a driving school, ensuring that vehicles are monitored and maintained efficiently is crucial for safety and operational effectiveness. Driving instructors and administrators require real-time access to vehicle data to make informed decisions, schedule maintenance, and ensure that vehicles are being used properly. Additionally, students benefit from the feedback provided by monitored data, helping them improve their driving skills.

This report outlines a comprehensive IoT solution designed for connected vehicles in a driving school fleet, focusing on real-time data collection, processing, and user interaction. By integrating various technologies such as Raspberry Pi, MQTT, Node-RED, and cloud services, we aim to create a robust system that not only monitors vehicle metrics but also provides valuable insights through an intuitive mobile application and a Telegram bot interface.

Our solution leverages modern IoT architecture to address common challenges faced by driving schools, including vehicle misuse, inefficient maintenance scheduling, and lack of real-time data accessibility. Through this system, we aim to enhance the overall safety, efficiency, and effectiveness of driving school operations.

The solution is scalable across multiple driving schools or multiple locations of the same driving school

Ideal Architecture

The ideal architecture for our IoT solution leverages cloud services to ensure scalability, reliability, and ease of management. By utilizing cloud platforms such as AWS (Amazon Web Services), we can efficiently handle the data processing, storage, and user interaction requirements of our system.

AWS provides a comprehensive suite of services that are well-suited for IoT applications. For our vehicle monitoring solution, we utilize several key AWS services, as we can also see in Figure 1:

- **AWS IoT Core**: This service allows us to connect our IoT devices (Raspberry Pi) securely to the cloud. AWS IoT Core handles device authentication, data ingestion, and real-time data processing, ensuring reliable communication between the vehicles and the cloud via various protocols such as MQTT and HTTPS.
- **Amazon RDS (Relational Database Service)**: For storing vehicle data, we use Amazon RDS, which provides a managed relational database service. This ensures high availability, automatic backups, and scalability for our SQL database, which stores historical vehicle metrics and other relevant data.
- **AWS Lambda**: AWS Lambda enables us to run serverless functions in response to events. We use Lambda functions to process incoming vehicle data, perform necessary computations, and trigger notifications or alerts.
- **Amazon S3 (Simple Storage Service)**: Amazon S3 is used for storing large amounts of data, such as logs and historical datasets, in a cost-effective and scalable manner.
- **Amazon SNS (Simple Notification Service)**: This service allows us to send notifications to users via SMS, email, or push notifications. It is integrated with our system to alert administrators about critical vehicle issues or maintenance reminders.
- **AWS QuickSight**: To provide powerful data visualization capabilities, we utilize AWS QuickSight. This service allows us to create and publish interactive dashboards that enable stakeholders to explore vehicle data visually, identify trends, and make data-driven decisions.

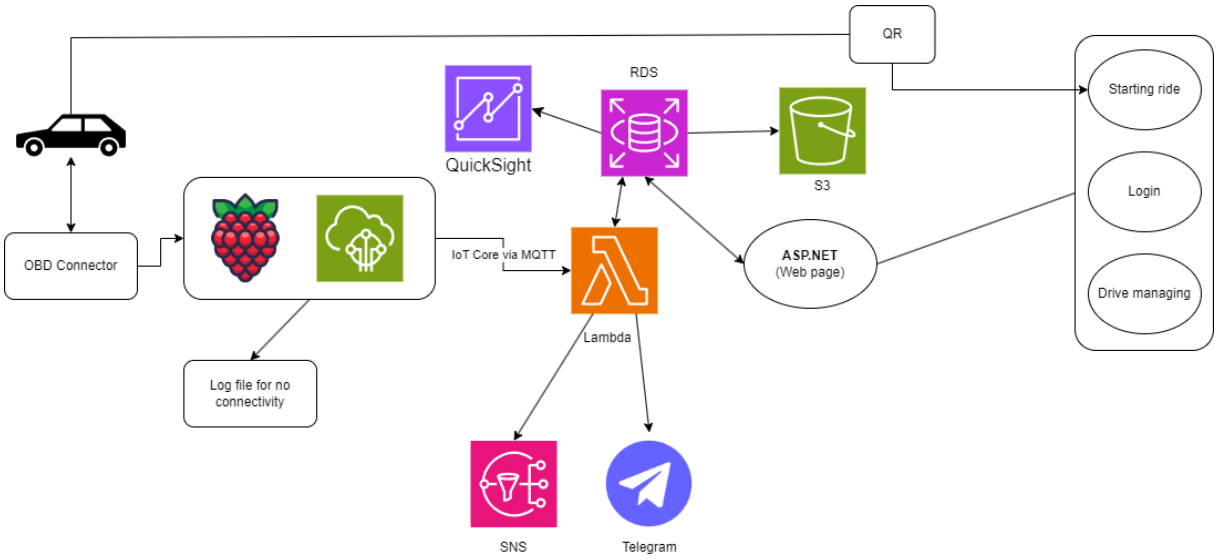

Figure 1: ideal architecture of IoCarT project using AWS

Data Flow and Processing

The data flow in our system is designed to ensure efficient processing and storage of vehicle data. The following steps outline the data flow from the vehicles to the cloud and back to the users:

1. **Data Collection**: Each vehicle is equipped with a Raspberry Pi device that collects OBD-II and GPS data. This data is published to an MQTT topic managed by AWS IoT Core.
2. **Data Ingestion**: AWS IoT Core receives the data from the vehicles and forwards it to an AWS Lambda function for processing.
3. **Data Processing**: The Lambda function processes the incoming data, performing tasks such as data validation, formatting, and computation of derived metrics (e.g., average speed, fuel consumption).
4. **Data Storage**: Processed data is then stored in Amazon RDS for structured querying and in Amazon S3 for long-term storage.
5. **User Interaction**: Users can access the data through a mobile application developed using ASP.NET Core and integrated with AWS services. The app fetches data from Amazon RDS and displays it in a user-friendly interface. Additionally, a Telegram bot allows users to interact with the system via predefined commands, retrieving real-time vehicle data and last known locations.
6. **Notifications**: Based on specific conditions (e.g., critical vehicle issues, maintenance needs), AWS Lambda triggers Amazon SNS to send notifications to administrators and relevant users.

Scalability and Reliability

By leveraging AWS services, our system can scale dynamically to accommodate an increasing number of vehicles and data volume. AWS IoT Core and Lambda provide automatic scaling capabilities, ensuring that our system remains responsive and efficient under varying loads. Furthermore, the use of managed services like Amazon RDS and S3 ensures high availability and durability of the stored data.

The cloud-based architecture also enhances the reliability of our solution. With AWS' global infrastructure, we can deploy our services in multiple regions, providing redundancy and minimizing downtime. Automated backups and disaster recovery mechanisms further ensure the continuity of our operations.

Security

Security is a critical aspect of our IoT solution. AWS IoT Core provides robust security features, including device authentication, secure communication (TLS), and fine-grained access control. Data at rest is encrypted using AWS KMS (Key Management Service), and data in transit is protected using secure communication protocols.

Access to the cloud resources is managed using AWS IAM (Identity and Access Management), ensuring that only authorized users and services can access sensitive data and perform critical operations. Regular security audits and compliance checks help maintain a secure and trustworthy system.

Demo Project

In our IoT solution for managing the driving school's fleet, certain components have been simulated to facilitate development and testing without requiring the full cloud infrastructure. This approach allows us to validate the system architecture and functionalities efficiently. Here, as we can see in Figure 2, we outline how we have simulated specific components and discuss the advantages and disadvantages of this approach.

Each vehicle in the demo setup is represented by a simulated edge device. Instead of using physical Raspberry Pi devices equipped with LTE and GPS hats connected to the OBD-II port, we use a Python script that mimics the behavior of these devices. This script generates and publishes OBD-II and GPS data to an MQTT topic, replicating the data collection process of the actual edge device.

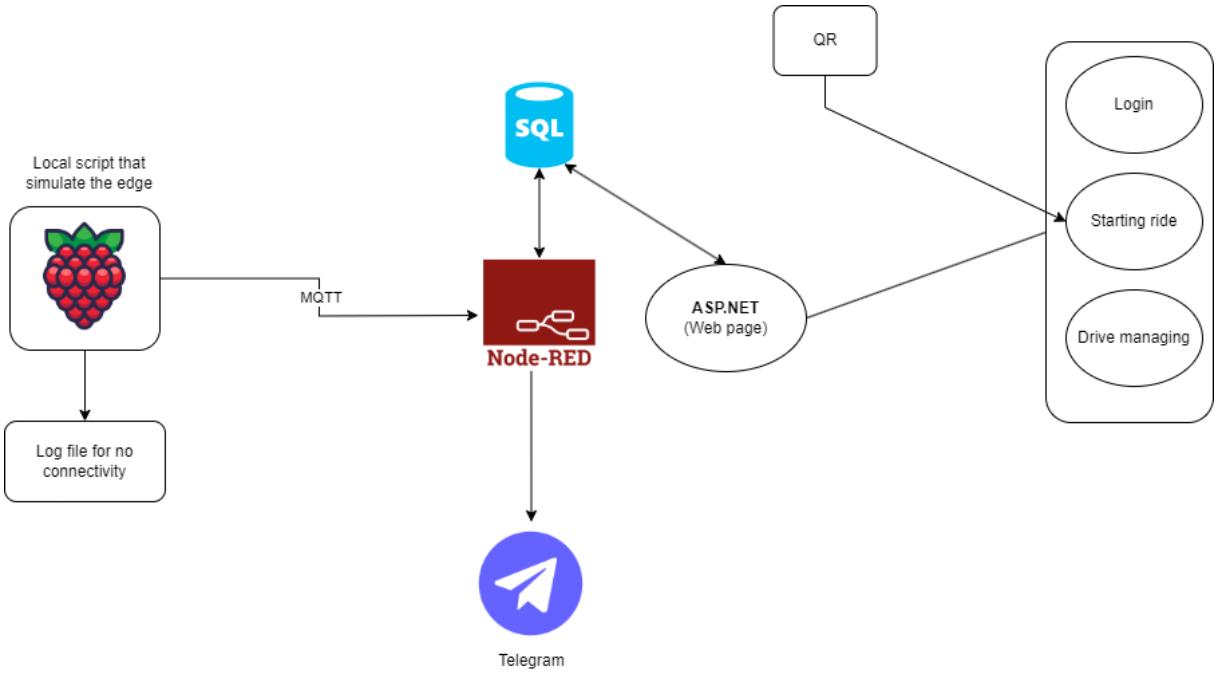

Figure 2: implemented architecture of IoCarT without cloud components

Edge Device Simulation

A Python script simulates the Raspberry Pi's data collection functionality. It generates random values for various vehicle parameters such as engine RPM, vehicle speed, throttle position, battery voltage, and GPS coordinates. This simulated data is published to an MQTT topic as if it were coming from the actual hardware.

Middleware Processing

Node-RED is used to process the data received from the MQTT broker. It parses the incoming data, temporarily stores it, and constructs SQL queries to persist the data in a SQL Server database hosted on a private server or cloud.

- **Advantages**:
  - Facilitates quick setup and iteration on data processing logic.
  - Visual programming in Node-RED simplifies the integration of various components.
- **Disadvantages**:
  - May not fully mimic the performance and behavior of a production-grade middleware solution.
  - Potential scalability issues when moving from simulation to real-world deployment.

The decision to simulate certain components of our IoT solution during the demo phase offers significant advantages in terms of speed, cost, and ease of development. However, it also comes with limitations that must be addressed before deploying the system in a real-world environment. By understanding these trade-offs, we can better prepare for transitioning from a simulated environment to a fully functional, cloud-based solution, ensuring that our system is robust, scalable, and reliable.

Implemented Components

With respect to the components just described, the application and the telegram bot part were implemented independently of the underlying architecture

In the next chapters we will describe each component more in detail.

Edge

In our IoT solution for managing a driving school's fleet, the edge component plays a crucial role. This component is typically embodied by a Raspberry Pi device installed in each vehicle, equipped with LTE and GPS capabilities, and connected to the vehicle's OBD-II port. The edge device is responsible for real-time data collection and transmission to the MQTT broker. However, in our experiment, the edge functionality is simulated to facilitate development and testing without requiring physical hardware.

The edge simulation is implemented using a Python script that mimics the behavior of the Raspberry Pi. This script generates and publishes OBD-II and GPS data to an MQTT topic, replicating the data collection process of the actual edge device.

The simulation script begins by configuring necessary parameters such as MQTT broker details, log files, and configuration files. It sets up logging to record data publishing activities and any errors that occur during execution.

OBD-II data is simulated using the *simulate_obd_data* function. This function generates random values for various vehicle parameters such as engine RPM, vehicle speed, throttle position, battery voltage, transmission temperature, oil temperature, oil pressure, fuel level, and fuel consumption rate. Diagnostic trouble codes (DTC) are also included to mimic real-world vehicle issues.

GPS data simulation is handled by the *GPS_Simulator* class, which reads predefined GPS coordinates from an Excel file. The *coordinate_generator* function yields one coordinate at a time, simulating the vehicle's movement over a route. This ensures that each iteration provides a new GPS position, replicating real-time location tracking.

Before publishing, the script augments the simulated OBD-II and GPS data with additional information such as the car ID, obtained from a configuration file. The data is then published to the MQTT broker under a specific topic associated with the vehicle's ID.

To ensure reliability, the script includes mechanisms for handling failed data transmissions. If the script encounters an error while publishing data, it saves the failed payloads to a local file. The *resend_failed_payloads* function attempts to resend these payloads in subsequent iterations, ensuring that data is eventually transmitted to the MQTT broker.

The script runs in an infinite loop, continuously generating, augmenting, and publishing data at regular intervals (every 2 seconds). This simulates the constant stream of data from the edge device to the cloud, providing a realistic test environment for the rest of the IoT solution.

Middleware architecture

The architecture of our middleware is designed to facilitate efficient communication and data processing for our IoT application, focusing on the integration of Node-RED, Telegram, and MQTT. We have selected these technologies for their robust capabilities and ease of integration, which collectively enable a seamless and responsive system for vehicle data monitoring and interaction.

Node-RED serves as the central hub of our middleware, orchestrating the data flow between various components. It is a powerful, flow-based development tool for visual programming, making it easy to wire together devices, APIs, and online services. In our system, Node-RED handles incoming data from the Raspberry Pi via MQTT, processes and stores this data in a SQL database, and facilitates user interactions through Telegram.

MQTT (Message Queuing Telemetry Transport) is employed for its lightweight and efficient messaging protocol, ideal for IoT applications. The Raspberry Pi, acting as the data source, publishes vehicle telemetry data to the MQTT broker. Node-RED subscribes to these MQTT topics, receiving data for further processing. This approach ensures reliable and real-time data transmission with minimal bandwidth usage, which is crucial for IoT environments where network resources may be constrained.

Upon receiving the data, Node-RED executes a series of functions to parse and store the information. The data, including vehicle speed, engine RPM, GPS coordinates, and other critical metrics, is parsed from the MQTT payload and stored in a global context within Node-RED. This data is then formatted into SQL queries and inserted into an MSSQL database. This structured storage allows for efficient querying and retrieval of historical data, supporting various analytical and reporting needs.

For user interaction, we integrate Telegram, a widely-used messaging platform, through the Node-RED Telegram nodes. Users can interact with our system using predefined commands such as /getData and /cars, which trigger specific functions within Node-RED. These commands allow users to retrieve the latest vehicle data or a list of registered vehicles. Node-RED processes these commands, fetches the necessary data from the global context or the database, and formats it into user-friendly messages sent back via Telegram. This interaction model provides users with an intuitive and accessible way to monitor vehicle data in real-time.

We chose Node-RED for its versatility and user-friendly interface, which significantly reduces development time and complexity. Its ability to integrate with various protocols and services out-of-the-box, such as MQTT and Telegram, makes it an ideal choice for our middleware. MQTT was selected for its efficiency in handling IoT communications, ensuring reliable data transfer with low overhead. Telegram was chosen for its popularity and ease of use, providing a convenient platform for user interaction.

App

The IoCarT app aims to streamline the process of managing and monitoring vehicle rides for driving instructors and students. Built using ASP.NET Core with Identity and Entity Framework, the application ensures robust user management and data persistence. Its primary goal is to provide a seamless interface for instructors to start and end rides, log ride errors, and view ride statistics, while also enabling administrators to manage users and ride bookings effectively.

The project utilizes ASP.NET Core Identity for secure user authentication, coupled with role-based access control to ensure that only authorized users can access specific functionalities. Sessions are used to maintain ride states and user messages, storing information such as ride status, error messages, and user roles.

Instructors have a straightforward workflow within the application. After logging in using their credentials, they can start a ride by scanning a QR code on the vehicle. The backend verifies if the instructor has a ride booked that can start at the current time. Upon successful validation, the ride status is updated, and the session reflects that the ride has started. During the ride, instructors can log specific ride errors through the application interface. Errors such as "Crossing", "Precedence", "Stop", and "Sidewalk" can be recorded by pressing corresponding buttons, which are then saved in the database and linked to the current ride. At the end of the ride, instructors press the "End Ride" button, updating the ride status to completed and logging the end time. Relevant statistics for the ride, including average speed, RPM, and fuel consumption, are computed and displayed in the section *Statistics*

Administrators have additional features available to them. They can book rides for instructors and students, using an admin interface that allows the selection of instructors and students, and scheduling of ride times. Admins can manage ride bookings, ensuring efficient scheduling and resource allocation. They also have the ability to assign roles to users, providing the right access levels, and manage roles by creating, updating, and deleting them as necessary.

As we can see in Figure 3 the user interface is designed for ease of use. The instructor's home page displays a prompt to scan a QR code to start a ride, along with buttons for logging ride errors and ending the ride. Dynamic interface elements are shown or hidden based on the ride status. The statistics page displays detailed ride statistics, providing a summary view of performance metrics, in addition to this the page also incorporates a mapping feature that visually represents the locations of logged ride errors. Using the Leaflet library in conjunction with marker clustering, the application displays error markers on an interactive map. Different error types are color-coded for easy identification. This geospatial visualization aids in analyzing error patterns and improving driving instruction by identifying common error locations.

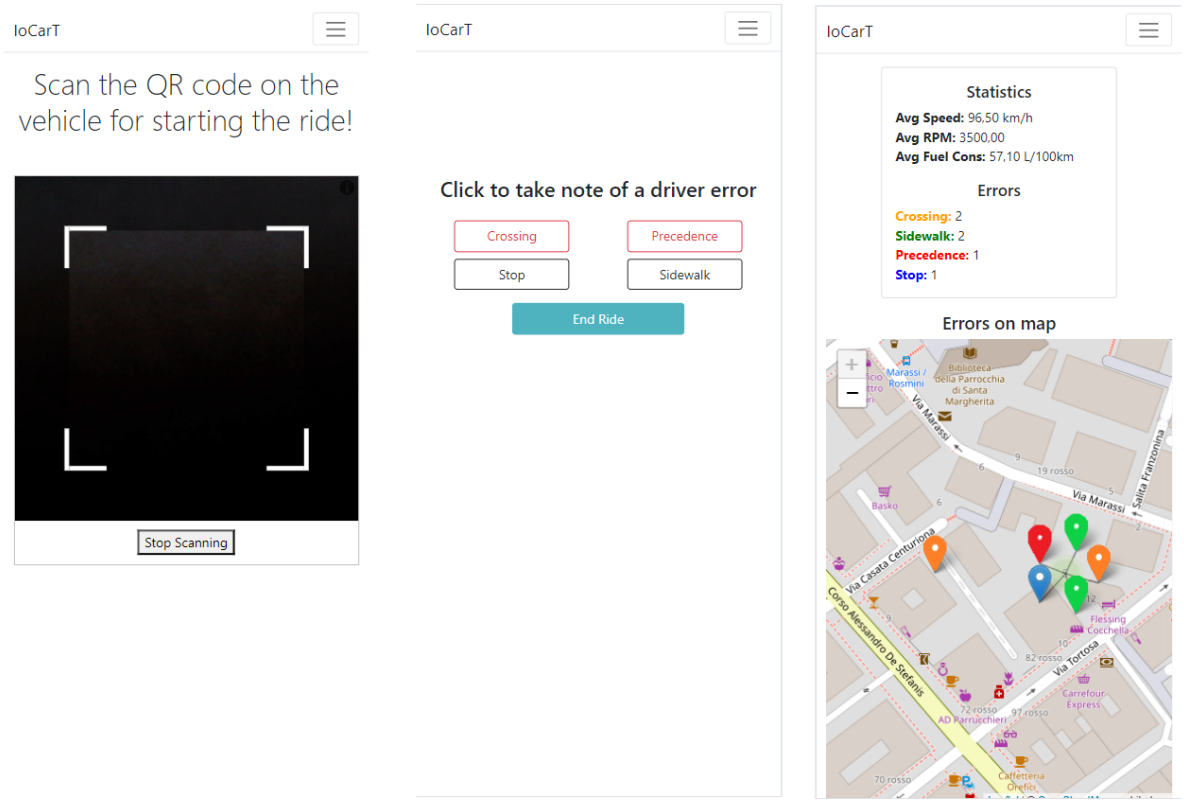

Figure 3: Main steps for starting the ride, logging errors and visualize the statistics

Technologically, the project leverages ASP.NET Core as the web framework, Entity Framework Core for data access and database operations, and ASP.NET Identity for user authentication and authorization. Session state is used to maintain user-specific data across requests, while JavaScript and jQuery enhance interactivity, handling QR code scanning and AJAX requests.

This approach ensures a consistent user experience across platforms, allowing instructors and students to access the app's functionalities directly from their mobile devices.