ECE:4890 Sr Elec and Computer Engineering Design
Final Project Documentation
4/29/2024
Team 3
Cole Arduser, Luke Farmer, Sam Loecke, Samuel Nicklaus

# 1. Introduction

**Rationale**

In today's era of automotive technology, there is an increasing need for a system that facilitates real-time access to vehicle diagnostics while also offering post-trip analytics. Current solutions utilizing the onboard diagnostics (OBD) port typically fall short in providing an all-in-one solution for live data access and post-process analysis. This project seeks to fill this gap by introducing a system that seamlessly integrates live vehicle data visualization in the car with an easy-to-use web application for detailed analytics and further processing after the trip.

**Goals and Objectives**

1. Design and implement a collection device to collect the relevant data from the vehicle's OBD 2 port. This device must relay data to a database through a web server.

2. Design and implement a database to store data collected from the vehicle during operation. This data will be used for historical data analysis in the web application.

3. Design and create a dashboard to show live engine data and engine codes in the vehicle while operating.

4. Design and create a web application that is used for viewing historical data collected in the car. The web application will be used to run advanced analytics and spot trends.

5. Design and implement a web server to facilitate communication between the collection system, the database, and the web application

# 2. Project Outcome

**Critical Assessment**

- Our team met all the objectives and requirements that we set out to accomplish.

**Summary of Deviations**

- We have made no significant deviations, except for the power source for the device in the car. We changed that to be more universal to all cars by using a battery to power the device.

**Team Member Contributions**

- Samuel Nicklaus: Created Server, Created Front End Web Client, Assisted on documentation, Assisted on emulator
- Cole Arduser: Worked on emulator, worked on hosting server application, assisted in server APIs
- Sam Loecke: Worked with OBD - Raspberry Pi connection, Backend for GUI, and Raspberry Pi database
- Luke Farmer: Procured Hardware and made GUI, Assisted in documentation

**Project Management**

- Bi-Weekly Check-Ins
    - We attended bi-weekly meetings with the professor and teaching assistant. This helped us assess if we were on schedule.
- Gantt Chart
    - We used a Gantt chart to track progress and tasks for the timeline of the project.
- Bug Tracking through GitHub
    - When a bug was found, we noted that in the GitHub repository to notify the other team members.
- Pull Requests through GitHub
    - Whenever a code change was pushed to the main branch, we made pull requests that had to be approved by at least two people before merging.
- Code Reviews
    - In addition to pull request approvals, we periodically completed code reviews to guarantee group knowledge and buy-in
- Overall Project Management Experience
    - We implemented multiple project management processes and built up other skills. All in all, we've learned the importance of planning when completing large projects. Good planning makes measuring progress much more manageable.

**Set Back Factors**

- Obtaining a Database
    - We were unable to obtain a database through the University of Iowa, so we had to pivot and find a free online database with limited storage. We ended up using Supabase as our DB option, which had a free tier.
- Difficulties with Kivy
    - Our software for developing a GUI is more difficult to use than expected, which changed how we are making our GUI look.
- Semester Workload
    - All of us have other classes going on right now, which has been a setback at times because we have to study for midterms or work on other projects at the same time.

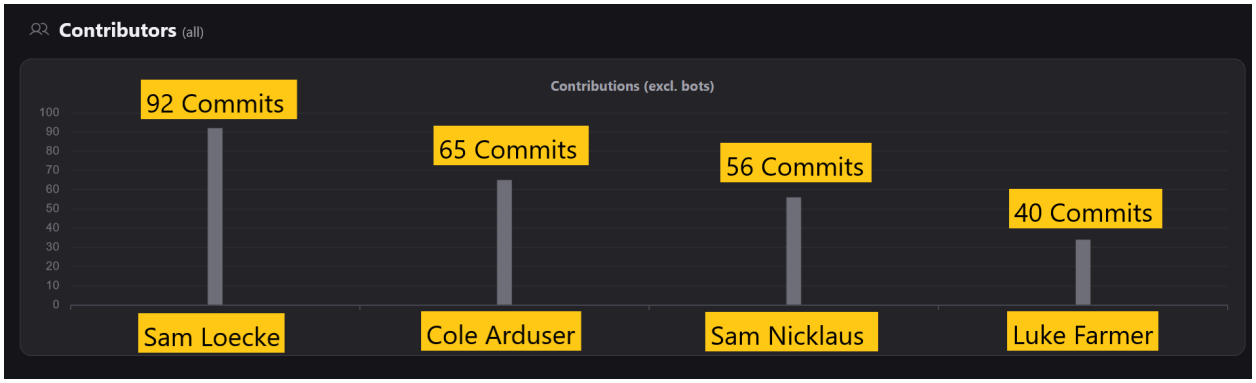


**Figure 1:** Team GitHub Contribution

## 3. Design Documentation

**Design Concept**

- We started by examining similar projects that integrate Bluetooth technology for communication between an OBD sensor and a Raspberry Pi with a display setup. Numerous open-source projects provided insights into the practical implementation of such systems. One notable reference was a Raspberry Pi-based OBD car project, which utilized Bluetooth for data transmission. This project served as a proof-of-concept, demonstrating the technical feasibility of our proposed system.
- Our research confirmed the compatibility of necessary libraries and software with the Raspberry Pi platform. We identified python-OBD, a comprehensive library for handling OBD data, which is compatible with Raspberry Pi and suitable for our project. For the web server and application, we drew upon our prior experience in creating a website interfaced with Raspberry Pi data, affirming the feasibility of integrating web connectivity and data management.
- We evaluated the hardware components required for our system, including the Raspberry Pi, Bluetooth OBD II adapter, and touchscreen display. Our selection criteria were based on processing capabilities, compatibility, and ease of integration.
- Finally, after our research, we determined there was a need for a system that is capable of onboard diagnosis with post-trip analytics. Our next step was to design a system to provide these capabilities.

**Design a solution that meets the specified needs**

- Our final design for the Smart OBD Diagnostic System (SODS) was designed to address the specific needs of car owners who seek a deeper understanding of their vehicle's performance and health. We tailored SODS to meet these needs through several key design features:
- Real-Time Diagnostic Data
    - A core need was the provision of real-time diagnostic data. SODS meets this by featuring a dashboard that offers instantaneous readings from the car's OBD II system. Parameters such as speed, RPM, and engine temperature are also logged for later analysis.
- Post-Trip Data Analysis
    - Post-trip analysis was an important requirement from our initial scope, aiming to provide users with a comprehensive look at their vehicle's performance over time. The SODS web application serves this need by allowing users to review their driving patterns, fuel efficiency trends, and recent trips.
- User-Friendly Interface
    - Recognizing that not all users are tech-savvy, we designed both the in-car dashboard and web application with a focus on user experience. The interfaces are intuitive, ensuring that users can easily navigate through various functions and understand their car data without a steep learning curve. This design consideration bridges the gap between complex vehicle diagnostics and user accessibility.

- Security and Data Protection
    - With growing concerns over data privacy, especially in connected systems, SODS incorporates encryption and secure data handling practices (such as the use of authentication tokens).
- Cost-Effectiveness
    - An emphasis was placed on creating a cost-effective solution. By utilizing open-source software and commonly available hardware, such as the Raspberry Pi, we have designed SODS to be affordable.

**Requirements**

L1.1: The system must collect data from the car's OBD II port
- L2.1.1: The system shall use a device capable of reading OBD II interface data to extract the available data from the OBD II port.
- L2.1.2: The data collection device shall have a Bluetooth interface for communicating data to a Bluetooth-enabled device.
- L2.1.3: The data collection device shall collect all available data from the OBD II port.
- L2.1.4: The data collection device shall automatically start collecting data when the car ignition is turned on and automatically stop when it is turned off.

L1.2: The system shall save collected data in a local database
- L2.2.1: The Bluetooth-enabled device receiving the data shall have a local storage database.
- L2.2.2: The local database shall store at least 48 driving hours of data at a maximum sampling rate before requiring synchronization with the web server.
- L2.2.3: The database shall implement efficient data storage formats.

L1.3: The user shall have a physical touchscreen interface for selecting and viewing live data in the car
- L2.3.1: The touchscreen interface shall display at least 720p resolution.
- L2.3.2: The interface shall have a menu for selecting from all available data points.
- L2.3.3: The interface shall display up to ten live data points simultaneously.
- L2.3.4: The touchscreen interface shall be no less than 7 inches in screen size.
- L2.3.5: The physical device shall have a controlled shutdown process when power is lost.

L1.4: The physical in-vehicle system shall display engine codes and alert the user of their severity
- L2.4.1: The system shall display a message for detected engine trouble codes.
- L2.4.2: The system shall display a severity assessment based on the error code.
- L2.4.3: Severity assessments shall be rated on a scale of 1 - 5.
- L2.4.4: The system shall recommend whether to stop immediately, stop when possible, continue with care, or continue as normal.

L1.5: The in-car system shall upload the collected data to a web server
- L2.5.1: The system shall upload data to a specified web server.

- L2.5.2: The system shall only attempt to upload data with an established internet connection.
- L2.5.3: The system shall only upload data collected since the last successful upload.

L1.6: The web server shall collect and store received information

- L2.6.1: The web server shall be always on and hosted online.
- L2.6.2: The web server shall receive data from the authorized in-vehicle device.
- L2.6.3: The web server shall only accept data from authorized devices.
- L2.6.4: The web server shall utilize an efficient database.

L1.7: The system shall include an efficient database for storing collected data

L1.8: The system shall include a web application for viewing, comparing, and analyzing collected data

- L2.8.1: The web application shall communicate with the web server for data retrieval.
- L2.8.2: The web application shall include options for viewing all available data points.
- L2.8.3: The application shall allow users to select a time period for graphing specific data points.
- L2.8.4: The web application shall have a homepage displaying common data points on page load.
- L2.8.5: The application shall allow users to compare and graph selected data points over a specified time range.

**Analysis of possible solutions and tradeoffs**

- We evaluated several microcontrollers and software frameworks before selecting Raspberry Pi and Flask, due to their balance of performance, community support, and cost.
- One significant tradeoff was the decision to utilize a web server-hosted database versus local storage to enhance remote accessibility while being mindful of data security and scalability.
- The chosen solution offers an optimal mix of real-time responsiveness, data management efficiency, and user accessibility. While alternative designs were capable, they did not match the robustness and scalability that our selected architecture provides.

**Constraints**

SC1: Size and Weight: The combined system shall fit within a maximum volume of 25 cm x 20 cm x 10 cm (L x W x H).
Measured device with ruler to determine that the volume of our Raspberry Pi head unit was under the dimensions listed above.

SC2: Power Consumption and Supply: The system shall operate on a standard 12V DC car power supply. The total power consumption of the system should not exceed 15 Watts under normal operating conditions.

Our system plugs into the cigarette lighter in the car and so uses 5V power that is powered from the 12V car battery. This draws a maximum of 2 amps meaning the overall power usage can not exceed 10 Watts. The system is designed for easy shutdown and startup to save power.

SC3: User Interface Response Time: The system shall respond to all user interface actions with a latency not exceeding 250 milliseconds.
Testing was performed in ideal server conditions, with the user going through GUI components with a stopwatch.

SC4: Data Transmission and Connectivity: The Bluetooth connection between the OBD II adapter and the Raspberry Pi shall maintain stable connectivity within a range of 5 meters, with a data transmission latency not exceeding 500 milliseconds under normal Conditions.
Connection between the OBD sensor and the Raspberry Pi was maintained beyond 10 feet of distance between the two. Queries were executed in under 10 milliseconds each. The amount of queries performed in a second was 42 and all in total took an average of around 100 milliseconds.

SC5: Operating Temperature: The system shall operate effectively in a temperature range from -10°C to 50°C (14°F to 122°F).
The system was tested under a wide range of temperatures throughout development. Operating temperature was around 10°F at the coldest and 80°F at the hottest. Hotter conditions were not tested, but all components are rated to work well beyond our specified temperature range

SC6: Durability and Vibration Resistance: The system shall withstand vibrations typical of a moving vehicle environment without malfunctioning or losing structural integrity.
The device was tested in a moving vehicle while operating, the user ensured no loss of functionality.

No additional constraints added.

**Standards**

- Vue.js v3.3.9: Adherence to ES6 for JavaScript and compliance with modern web development standards including HTML5 and CSS3.
- Python v3.12.0 & Flask v3.0.x & Kivy 2.2.1: Using the latest stable release for security and performance.
- PostgreSQL v16.1: Using best practice with data protection when handling and storing user car data.
- SQLite v3.44.2
- RESTful API Design
- OBD-II Protocols: Compliance with OBD-II standards for vehicle diagnostics and data retrieval.
- Bluetooth: Adherence to Bluetooth Core Specification for wireless data transmission.

- Wi-Fi: Compliance with IEEE 802.11
- Compliance with safety standards for electronic devices to prevent hazards such as overheating or electrical malfunctions

For our final product, we discussed and implemented our own login code, as opposed to the Google OAuth 2.0 we first discussed as a standard in the Final Project Proposal. Overall, this was a design decision and did not affect the functionality of our web application. All other standards from the project proposal were followed.

No additional standards were added.

**Architecture**

| Item No. | Part Description | Quantity | Supplier | Notes |
|---|---|---|---|---|
| 1 | Raspberry Pi 4 Model B | 1 | Raspberry Pi | Central processing unit |
| 2 | OBD II Bluetooth Adapter | 1 | Veepeak | Vehicle data retrieval |
| 3 | 7-inch Touchscreen Display | 1 | Raspberry Pi | User interface display |
| 4 | Micro SD Card 32GB | 1 | SanDisk | Storage for Raspberry Pi OS |
| 5 | Raspberry Pi Case | 1 | Raspberry Pi | Enclosure for Raspberry Pi |
| 6 | Micro HDMI to HDMI Cable | 1 | Various | Connect Pi to Touchscreen |
| 7 | GPIO Ribbon Cable | 1 | Adafruit | For internal connections |

**Figure 2:** Parts list for our project


**Figure 3:** Bluetooth OBD Sensor that is plugged into the car

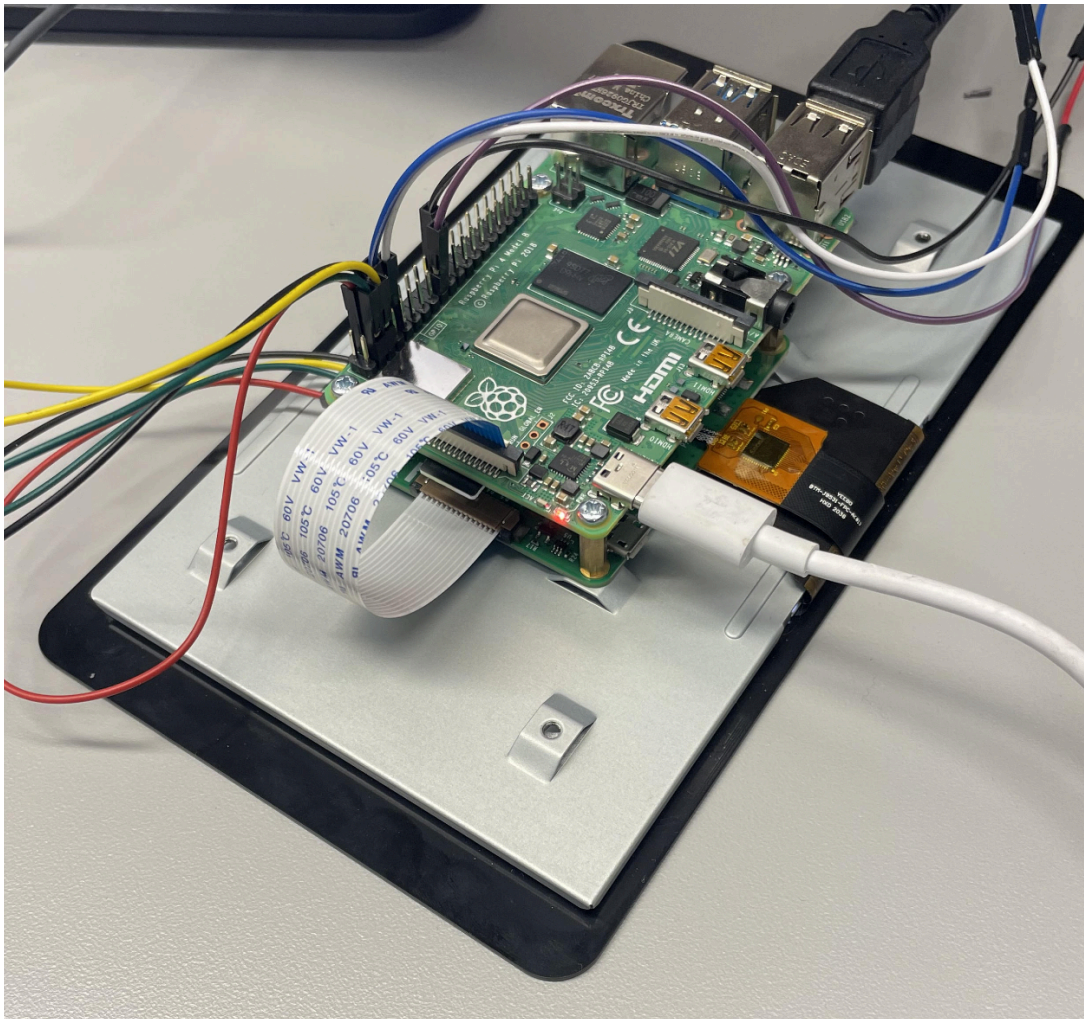

**Figure 4:** Backside of the 7” touchscreen with the Pi attached

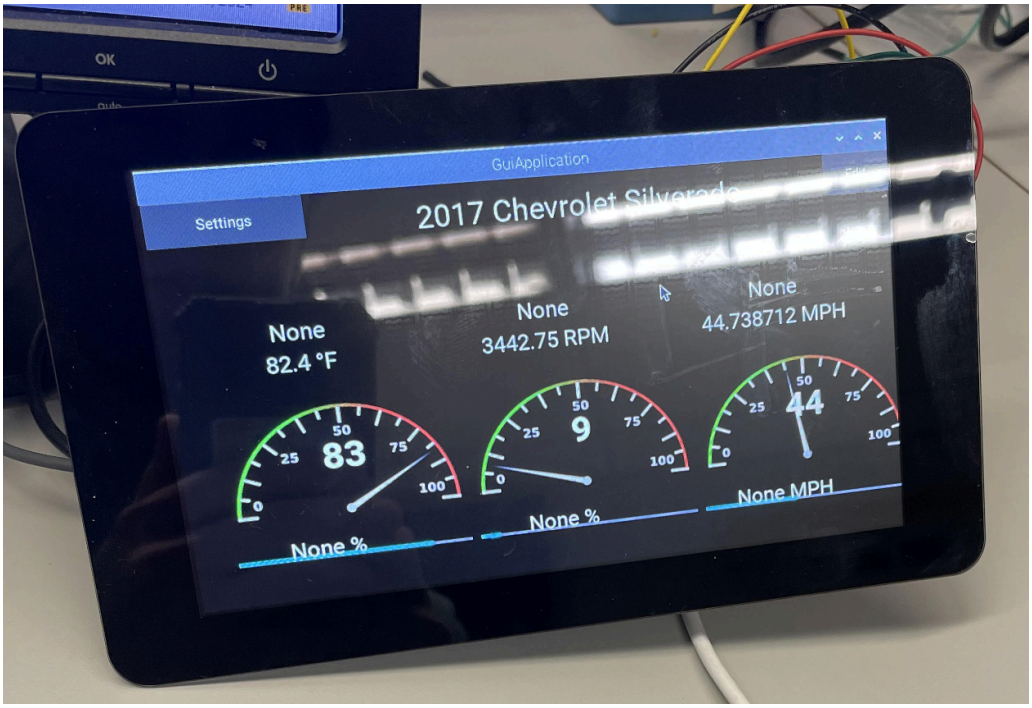


**Figure 5:** Front of 7” touchscreen showing main screen of the GUI

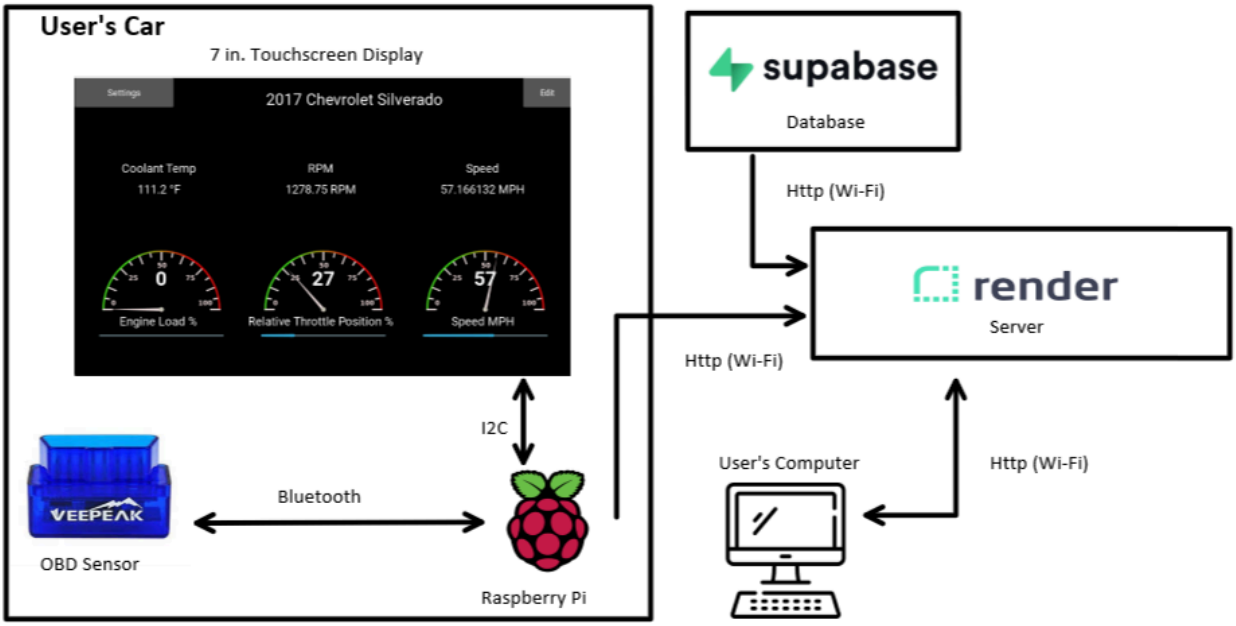


**Figure 6: System Overview Diagram**

**OBD Sensor:**

The Bluetooth OBD sensor we used for our project is the Veepeak OBD II Scanner. This sensor directly plugs into the user’s OBD port and is used to read all available data from the user’s car. The Bluetooth capabilities of the sensor allow us to communicate with the sensor using the Raspberry Pi. When the program on the Pi is started, it tries to make a Bluetooth connection to the sensor. Once the connection is established, the Pi then sends commands to

the sensor. The sensor takes command, runs queries on the OBD port for the requested data, and then returns the data over Bluetooth to the Pi. The sensor is compatible with all gasoline cars made after 1996. The sensor is powered on and off with the ignition of the car.

**Raspberry Pi:**

The Raspberry Pi we are using to facilitate the collection of data from the sensor is a model 4 B. The Pi uses its Bluetooth capabilities to connect to the sensor and collect data from it. The Pi gives queries for specific data points to the sensor and waits for a response back. The Pi uses the Python OBD library (https://python-obd.readthedocs.io/en/latest/) to facilitate the querying of the sensor. The library assists in connecting to the sensor and making accurate queries for different data points. The library supports a wide range of different data points and functions.

When the program on the Pi is started, it queries the sensor to see all the available commands the car's OBD port has. After this, the Pi sends queries for all available data points every second. The data is collected and stored in the local MariaDB database (https://mariadb.org/) on the Pi. This database is a MySQL-based database specifically for the Raspberry Pi. The Pi also stores the data in a dictionary to be able to give it to the live data dashboard GUI when it requests it. The GUI and data collection programs are multithreaded to be able to collect data while managing GUI events. We will discuss more about the GUI in the following section. The program is set up to run when the Pi is powered on. This is meant to keep the user from accessing the other functions of the Pi. The program is also set to automatically shut off when the car's RPMs drop below 100. This indicates that the car was shut off and the Pi can cease collection.

When the user selects the upload button on the GUI, if the Pi is connected to the internet, it sends all the data in its local database to the server. If a positive confirmation is received indicating that the server successfully received the data, all the data in the local database is cleared. The Pi connects to the server via HTTP calls and uses Python HTTP requests. The server will then process the received data. The Pi sends only the data points it collected and does not do any additional processing, so the heavy processing is on the server side.

**OBD Emulator:**

We used an OBD Emulator, specifically leveraging the Python ELM327 emulator (https://github.com/Ircama/ELM327-emulator), to simulate car diagnostics data without the need to physically operate a vehicle. This approach was instrumental for several reasons. It allowed us to conduct tests and refine our system under a controlled environment, ensuring consistent data availability. It also allowed us to generate specific conditions and data that may be hard to replicate inside a real car in real life.

The ELM327 emulator interacts with the main python-OBD library (https://github.com/brendan-w/python-OBD), which is used by the PI to decode data from the actual OBD sensor. The emulator mimics the data transmission one would expect from an actual vehicle's OBD II port. To facilitate this interaction on Windows machines, we utilized com0com, a kernel-mode, virtual serial port emulator, which enabled us to assign the emulator to one COM port and the script for the Raspberry Pi to another, effectively looping the

communication back to simulate a real data flow between the OBD II sensor and our system. This allowed us to run the emulator and the Pi script from the same computer.

To ensure the emulator provided realistic vehicle data, we forked the original ELM327 emulator project and customized the data responses to more accurately reflect real-world driving conditions. Modifications were made to various parameters such as miles per hour (MPH), revolutions per minute (RPM), oil and coolant temperatures, runtime, and Diagnostic Trouble Codes (DTCs). These adjustments were critical in testing edge cases as well as the data handling on the server side.

Through these customizations, the emulator became an essential tool in our development process. It allows us to efficiently simulate numerous driving scenarios and diagnostic alerts. It accelerated our development cycle by enabling rapid prototyping and testing.

**Touchscreen Display and GUI:**

The touchscreen will be used to display the relevant data in the car and allow the user to select what data they would like to see displayed on the main screen. We use a 7” Raspberry Pi touchscreen as our display.

For the GUI displayed on the touchscreen, we have three main screens: Display, Edit Selections, and Upload Data. We use Kivy and KivyMD to create an appealing and easy to use GUI for our users. We will now discuss in more detail each screen of the GUI.

Figure 13 shows the main display screen. The gauges are custom components that move to give the user a more interactive display. The top 3 data displays are text only, which gives the user the ability to select data values that don’t make sense to be displayed with gauges.

Figure 14 shows the upload data screen. The user can select “Add New Wi-Fi Network” which brings up a pop-up where the user can add a Wi-Fi network. The Wi-Fi status is displayed on this screen too. If the device is connected to Wi-Fi, the user is able to upload the local Raspberry Pi data to the server.

Figure 15 shows the edit screen. On this screen, the user can select drop-down buttons labeled “Select Data Point.” When that is selected, a drop-down displaying all valid data point values opens and the user is able to select what they would like displayed at each of the 6 values.

**Server:**

The server is built using Python and the Flask library for receiving and facilitating data between the Raspberry Pi local client, the database, and the web client. The server application contains a series of API routes that the clients can access for sending and receiving important data. The specific API requests are sent over HTTP and require a valid encrypted token to be passed to the server in order for the route function to complete. This token is created and handled by the JWT library and ensures unauthorized users cannot access the server functions. The following routes and their descriptions are listed below:

Verify: Function that takes in a user token and returns a 200 code if the token is valid, otherwise return a 500 error code.

Login: Function that takes in a username and password and returns a valid token to the user if the data matches what's in the database, otherwise returns an error 500 code.

Grab Car Details: If the user is verified, return details about the users' car such as make, model, and year manufactured.

Post Car Details: If the user is verified, inserts details about the users' car in the database such as make, model, and year manufactured.

Grab OBD Data: If the user is verified, returns all current error codes associated with the users' vehicle along with a description of the code, possible causes, and possible symptoms experienced by the error.

Post DTC Data: If the request is from the Raspberry Pi, insert current vehicle error codes about a users' vehicle into the database along with information about the error.

Grab Notifications: If the user is verified, return all unread notifications to the user so they can view them in the web client. Information returned could include car error codes, general information about their vehicle, or updates from the developers.

Dismiss Notifications: If the user is verified and provides a valid Notification ID, the notification in the database is dismissed so they don’t receive it again on refresh.

Staging Function: If the request is from the Raspberry Pi, process all incoming car data and store it in the database. The data sent to the server is a list of car data containing a snapshot of the car's runtime, speed, coolant temperature, oil temperature, airflow rate, and more for a given second in time. The server condenses this information down to daily averages (average speed, rpm, temperatures, etc.) along with calculated miles per gallon (MPG) value and inserts it into the database.
This function also takes the data and splits the list up into sets of “trips”. A trip starts when the user starts driving and ends when the time between drives is 30 minutes or more. Each trip is then processed individually and stored in the database, storing information such as start time, end time, average MPG, runtime, average engine load, starting and ending latitude points, and starting and ending longitude points.

Grab Data: If the user is verified, return a list of daily summarized driving data

Grab Trips: If the user is verified, return a list of all trips in the database

Grab Current Data: If the user is verified, returns the driving data for the last time they drove the car, returning information such as speed, fuel efficiency, runtime, etc.

Grab Current Trip: If the user is verified, and the user provides a trip ID, return all information associated with that trip.

Grab Graph Data: If the user is verified, return all driving data from the last month.

**Database:**
As mentioned before in the section on the Raspberry Pi, the database used by the Pi to store live data is a MariaDB database. It is based on MySQL and is specifically made to run on the smaller system of a Raspberry Pi. The data is stored in the database once every second as the system collects it. All the available data points from the car's OBD port are stored. There is just one table in the database called VehicleData which has columns for all the possible data points that can be collected from an OBD port. Each second, one entry is made into the database with all data points included. Once the user requests to upload the data to the server, all the information in the table is sent to the server. Once the Pi receives confirmation that the server received its data, the data in the VehicleData table is cleared.

Our server database is hosted using Supabase, a free, online, database hosting platform. Our database is based on PostgreSQL and uses relational tables to grab and insert values. Through Supabase we use an authorization key, so outside users cannot edit the database. This key is distributed only to the Server, so it is the only entity that can access database information. Below is a description of each table and what they do:

Users: Contains username and password to log in.

Car: Contains information about the user's car, including make, model, and year. Car_id is used as a key for table OBD.

OBD: Contains error information related to a specific car. The car is linked to the table by the car_id. Specific error information includes the error code, code description, common symptoms of the code, common causes of the code, and a status boolean that tells us if the user has read the error notification that gets sent on the web client.

DrivingData: Contains summarized information about a user's driving habits for a given day. Information includes number of entries, which is used to average out the other data, a timestamp, average speed (MPH), total runtime of the vehicle (Seconds), average coolant temperature (Fahrenheit), average oil temperature (Fahrenheit), and average miles per gallon.

Trips: Contains summarized information about a user's driving habits for a given trip. Information includes start time, end time, start latitude and longitude, end latitude and longitude, along with all the normal averages in the previous table entry.

Below is our ER Diagram that shows the relationships between tables, please note that "PiData" is all the information stored locally in the head unit and not in Supabase.

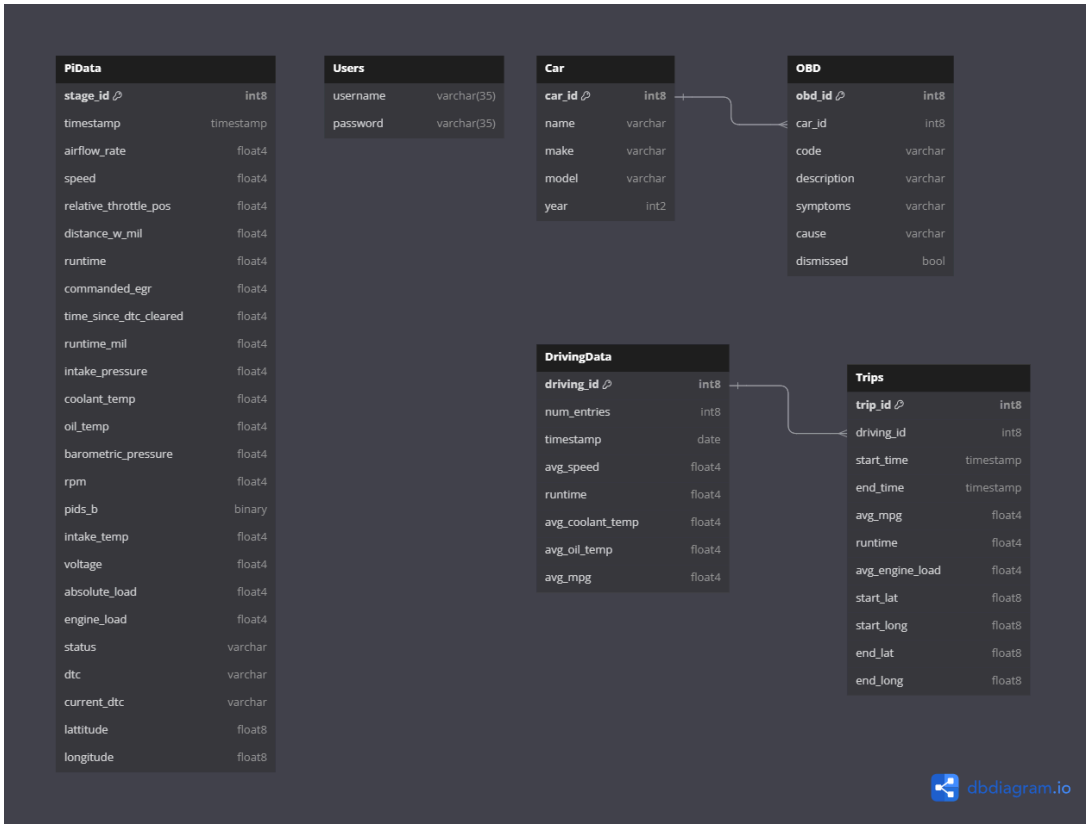


**Figure 7:** ER-Diagram of database

**Web Client**

The web client is used to help summarize and display historical car data to the user, along with information about possible OBD error codes the car currently has active. The client was built using Vue.js, a JavaScript framework that combines CSS and HTML into a single reactive component. The following libraries were used to enhance the user experience:

Leaflet: Map API Library that allows us to display route info to the user.
Vue Cookies: Allows the client to store the user token so they do not need to log in every time they refresh the browser.
Vuetify: Component Library for Vue.js
V-Calendar: Reactive calendar component that allows users to view daily average data.
Axios: Allows web clients to securely send and receive data from the server.

At the top of the web application, we have a permanent navigation bar that provides a direct link to login/logout, along with a notification bell that shows important information to the user (such as if their car has an error code).

Figure 8 is our login screen. On this screen, the user has the option to fill in a username and password to login. If the user enters information into the text boxes and presses the button, an API call is made to the server to check if the information entered is correct. If the information is

correct, a user token is stored by the client, and the user is redirected to the home page. Otherwise, they are prompted to try again.
Figure 9 is our home screen. On this screen, a summarized view of important car data is displayed to the user. In the top left we have a calendar that shows the last month of summarized stats about the car (such as MPG, runtime, etc.) when clicked. In the top right of the screen, we have the most current-day stats of the car, which includes similar metrics at a glance. At the bottom of the screen we have a list of the user's most recent trips, when clicked, the user is redirected to the trips page.

Additional information about the UI can be found below in the **UI/UX** Section.

**UI/UX**

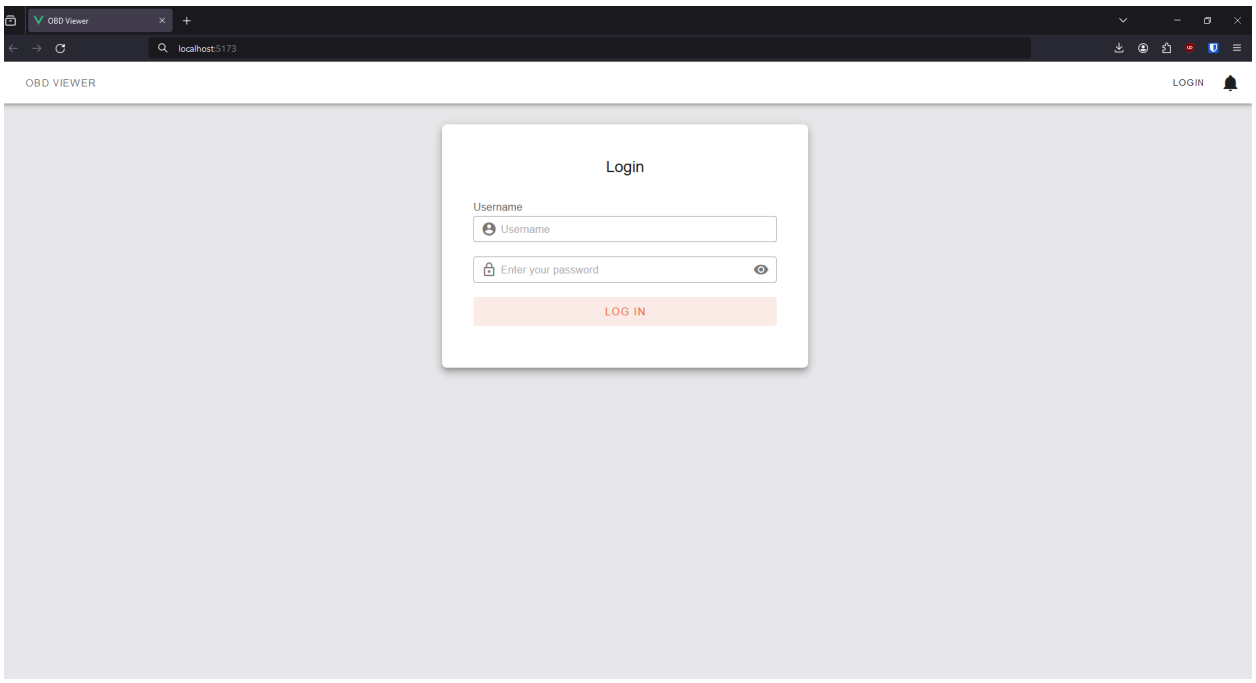


**Figure 8:** Web Client Login screen, designed for the user to enter a username and password to access their car data

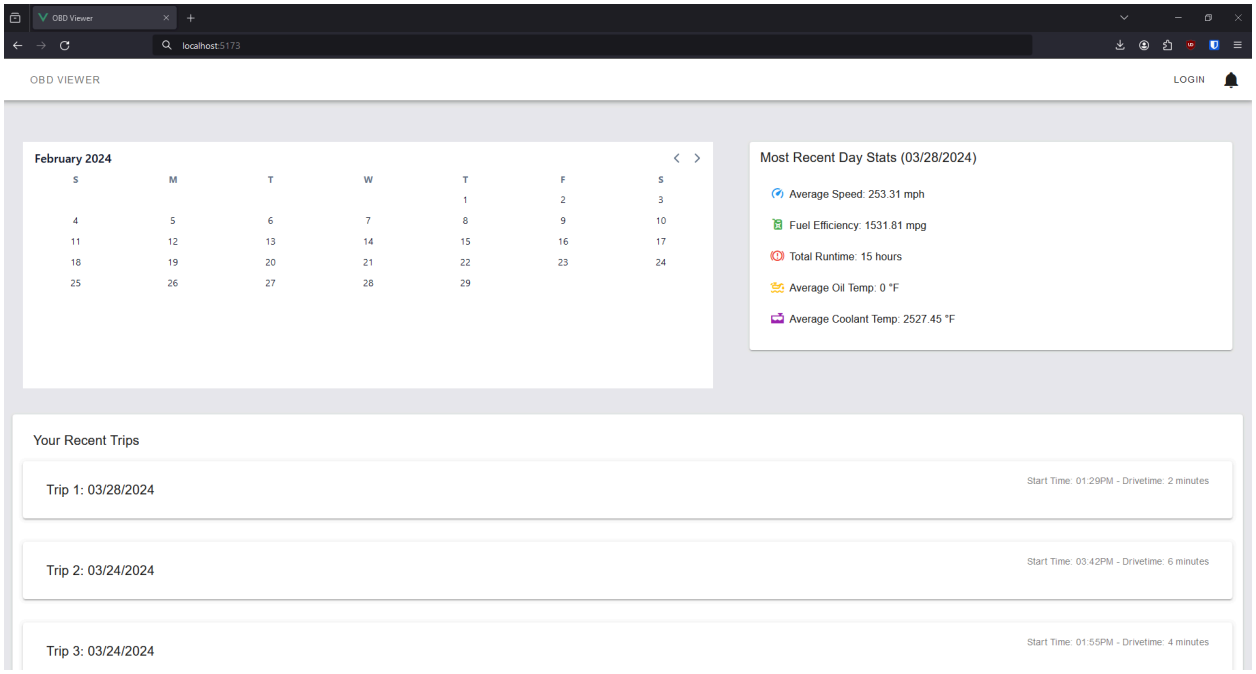


**Figure 9:** Home screen of the web client, features include calendar view for daily stats (top left), most recent drive stats (top right), and list of most recently completed trips (bottom)

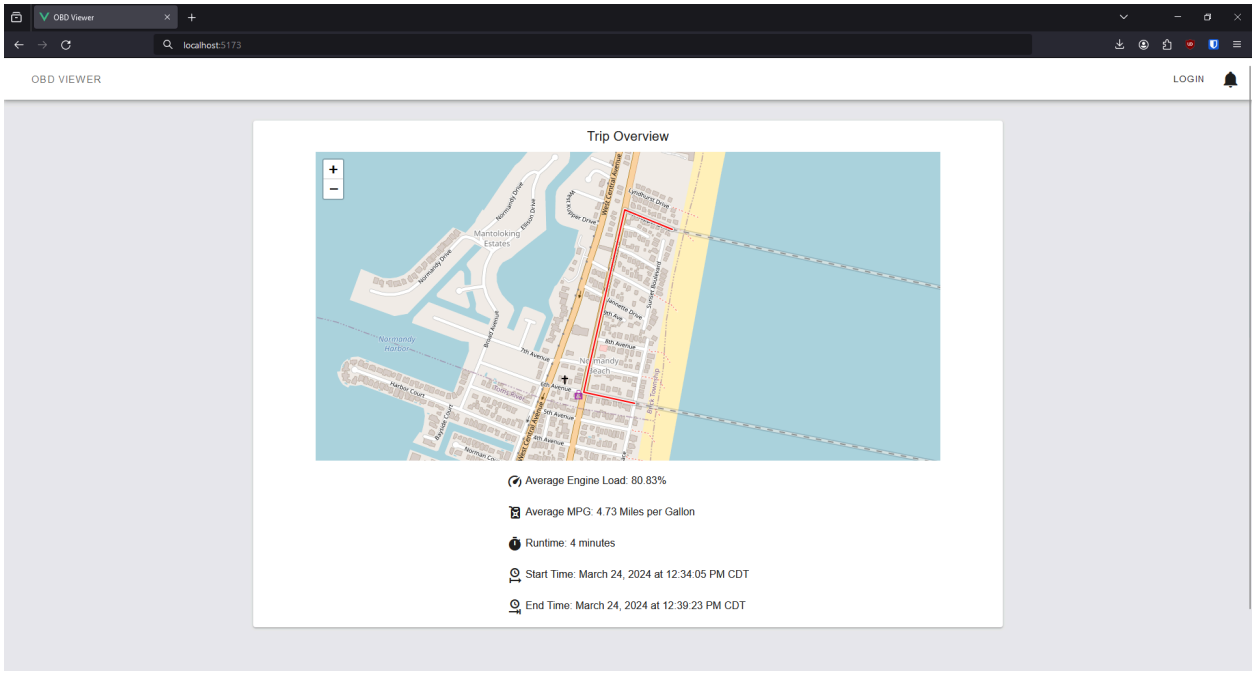


**Figure 10:** Trip overview page, showing the user trip data such as average engine load, MPG, and runtime. Also maps out the user's route they took

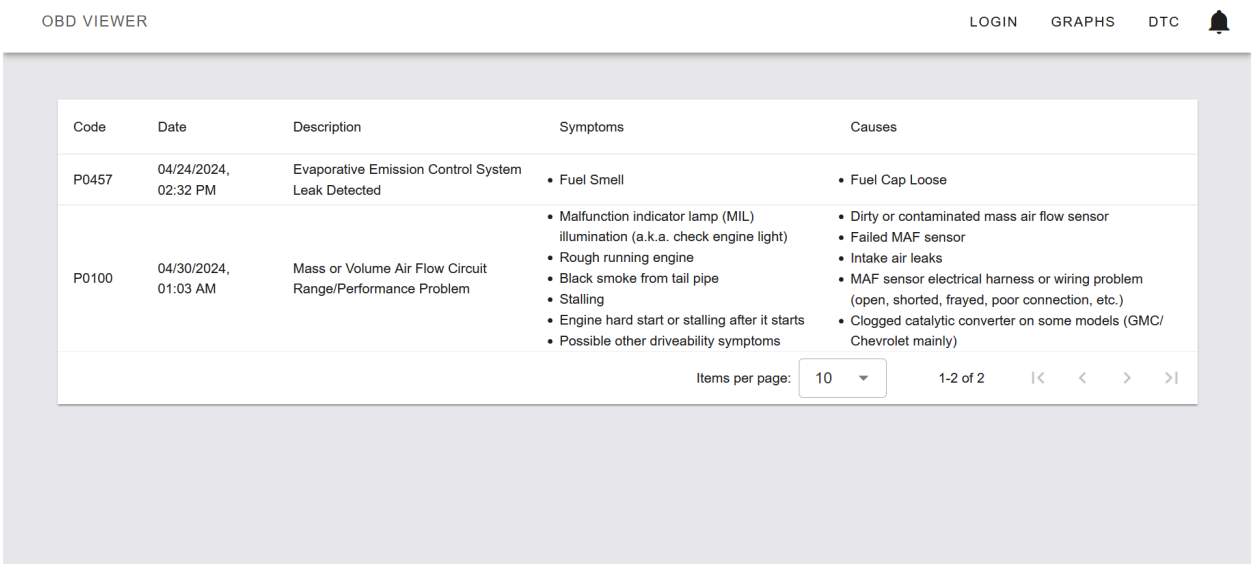


**Figure 11:** DTC code view page, showing a history of trouble codes and possible causes/symptoms

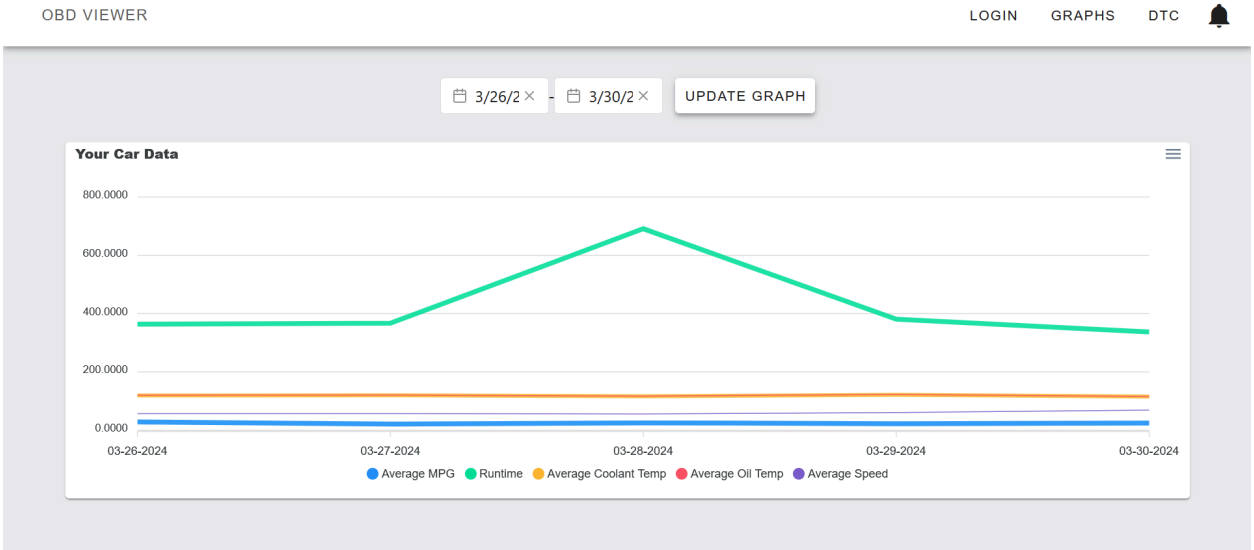


**Figure 12:** Historical overview page, allowing users to view data between any two sets of dates

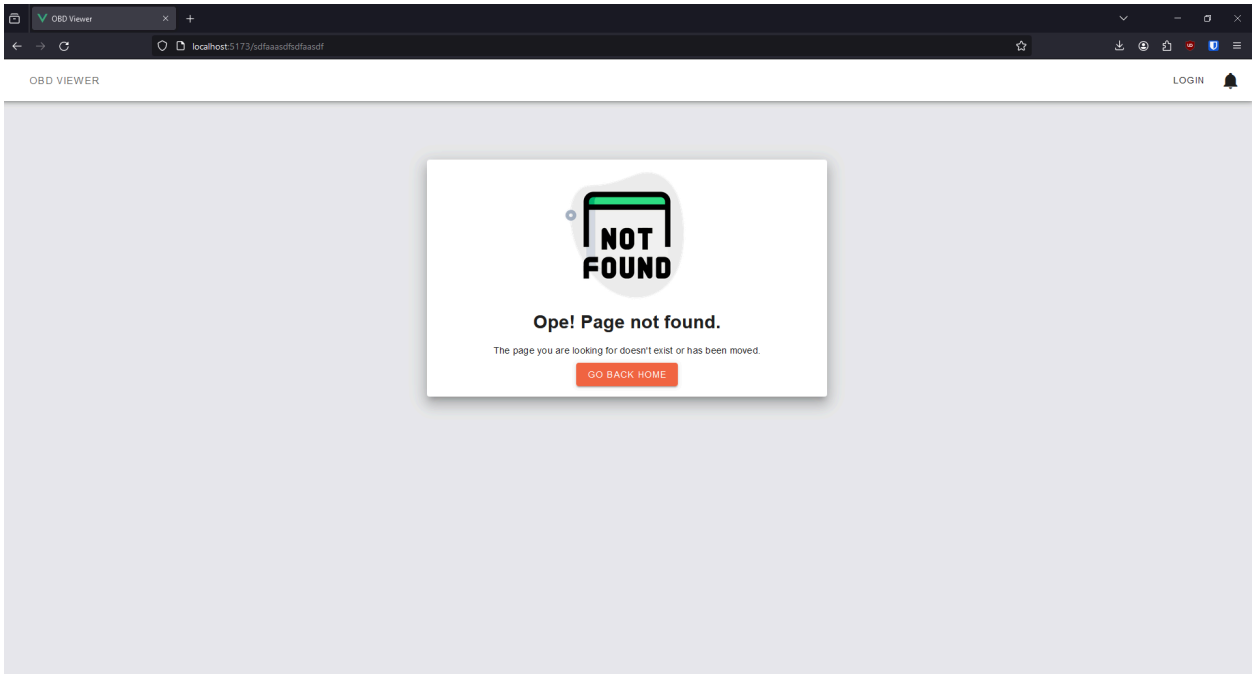


**Figure 13:** 404 Page, displayed to the user when an incorrect route has been entered in the address bar

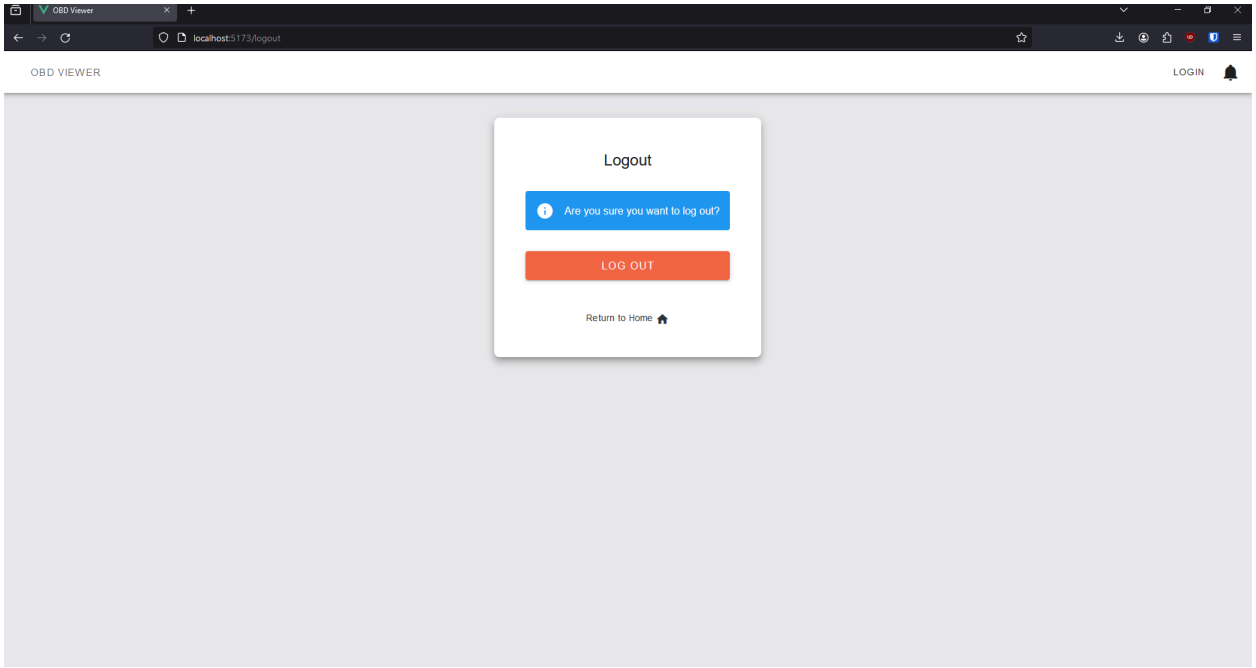


**Figure 14:** Logout page for the user

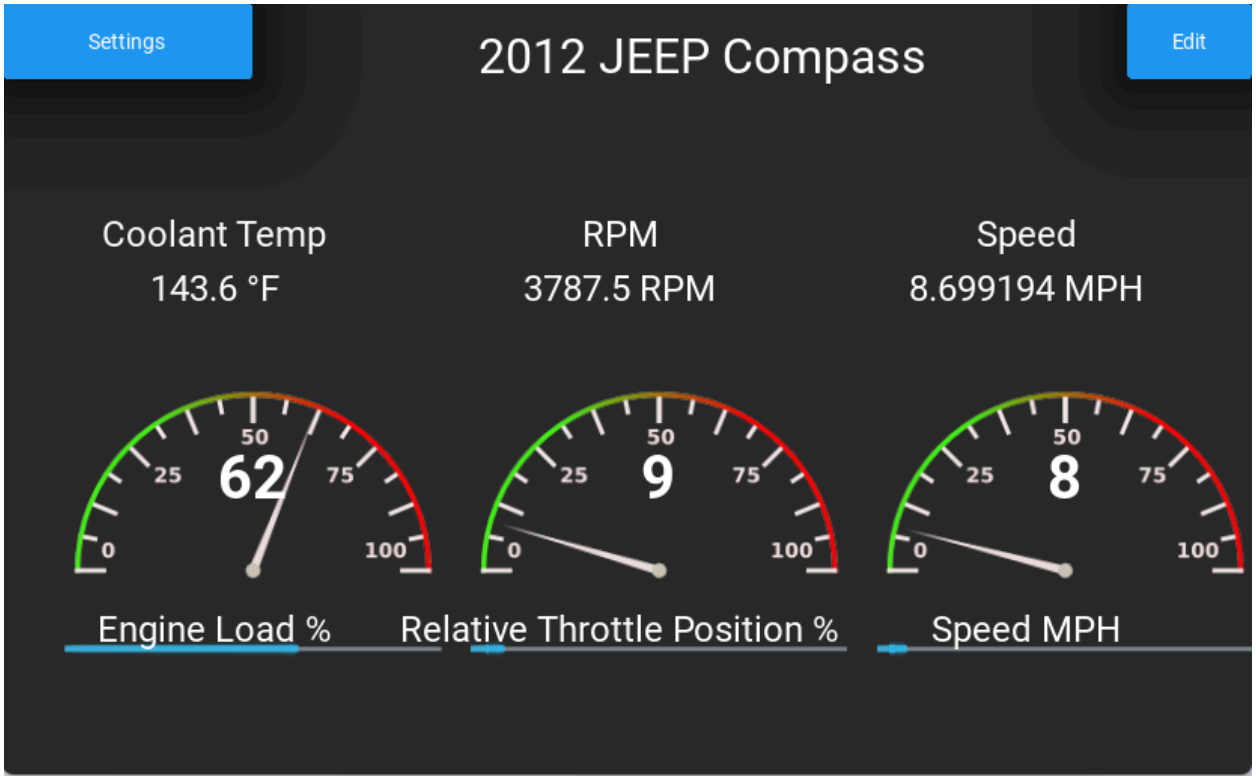


**Figure 15:** This is the home screen for the touchscreen GUI where the data is displayed

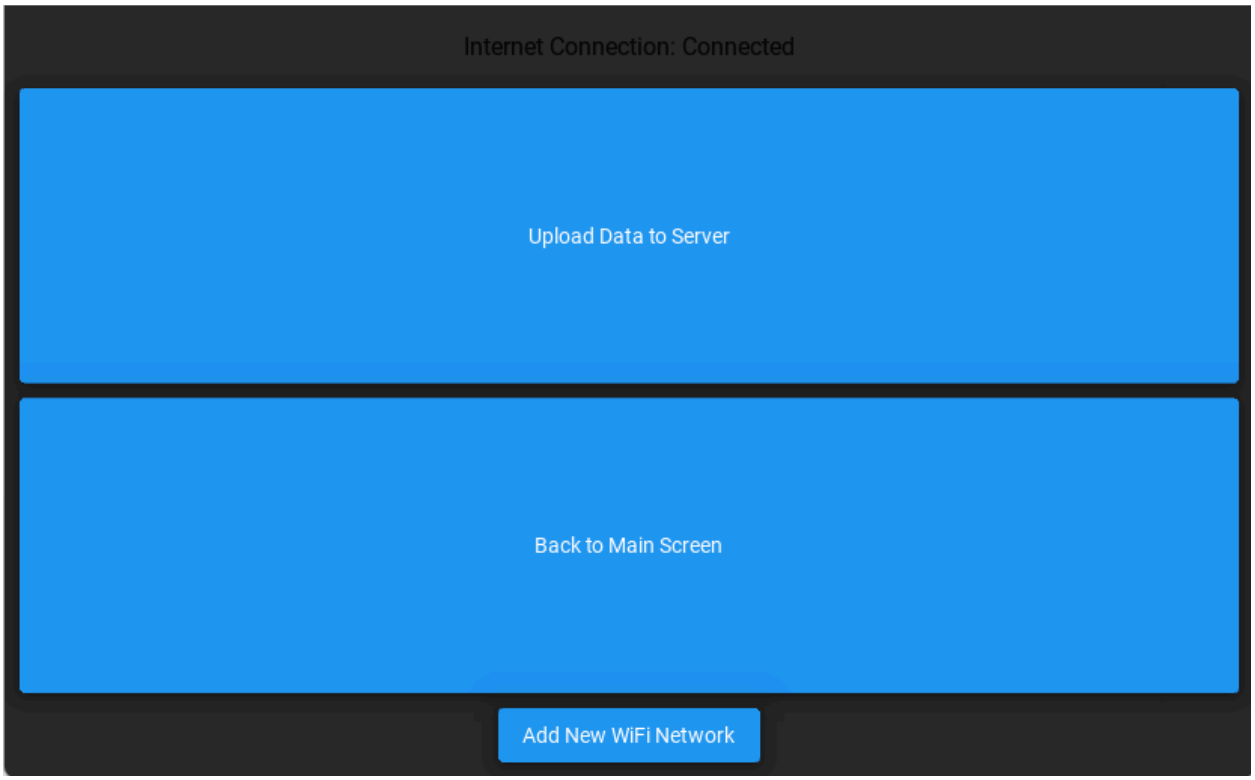


**Figure 16:** This is the screen the user uses to upload data to the server. They are able to add a Wi-Fi network, upload data to the server, and move back to the main screen.

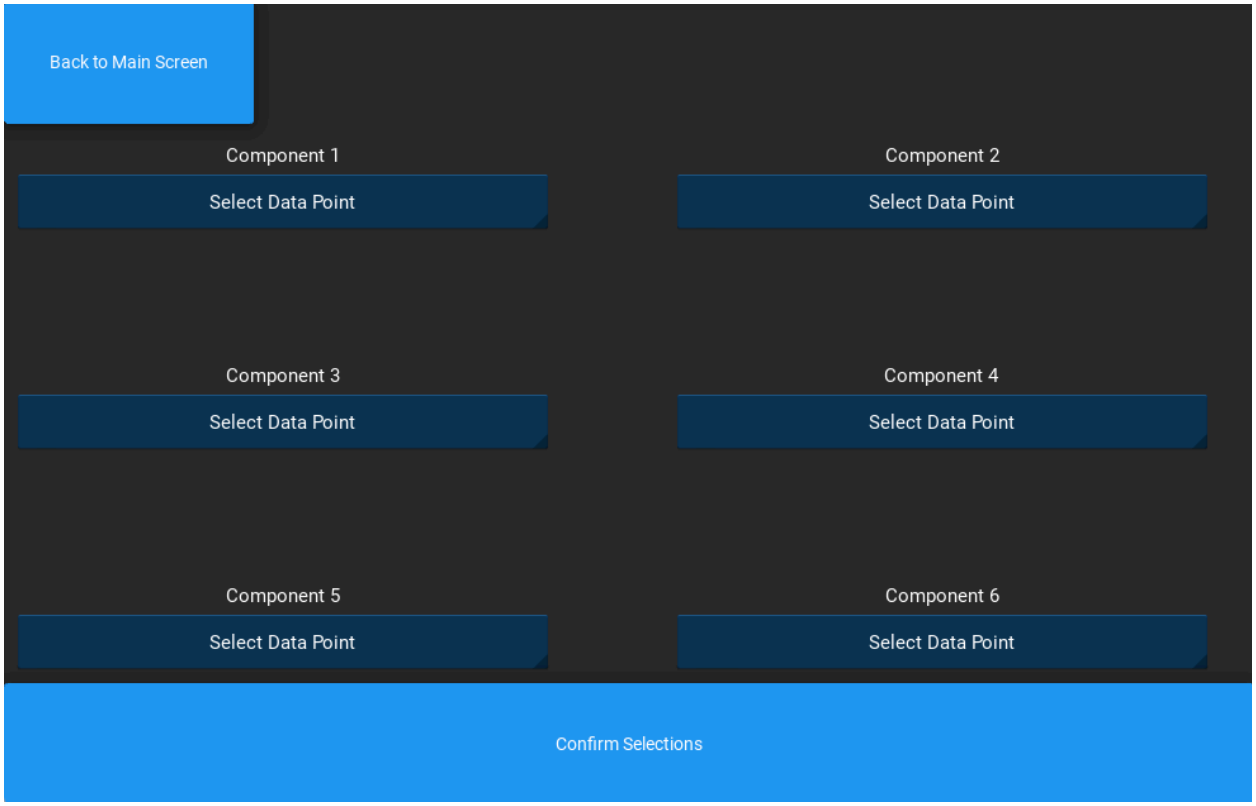


**Figure 17:** This is the screen where the uses select what data they would like to see displayed on the home screen.

**Maintenance**

Dependencies of our system include Supabase, which is a free, online database hosting service that we use to store trip and vehicle data. This service is free until the 0.5MB database storage limit is reached, then it turns into a paid service. Wi-Fi network access for the local client is also required to upload data to the server. This connection does not have to permanently exist, but ideally once a day there needs to be a connection.

To host the API endpoints that allow the frontend and PI to access data, we used Render. Render offers a limited, free tier, providing a simple and easy solution for projects like ours. This platform supports our objective of ensuring reliable data transfer between the Pi devices and our system but also simplifies the deployment process. One of the standout features they offer is the ability to deploy our server-side applications directly from a specific GitHub branch and commit, streamlining updates and maintenance. This direct integration with GitHub, combined with Render's scalability and ease of use, ensures our system remains responsive and up-to-date, facilitating smooth data synchronization across devices without hassle.

As for limitations, as talked about before, the database service is storage-limited, which means eventually the database will either fill up or the user will have to pay for the service. Render’s free tier also has limitations. Some examples are: the amount of RAM is limited to 512MB, the service is spun down after prolonged inactivity, and there is no support for SSH,

jobs, or persistent disks. Possible enhancements of our product revolve around scalability, the device was made for a single car

Other possible enhancements that in the future could focus on more advanced analytics on the collected data. Allowing users to create customized functions derived from the collected data points would allow them to fulfill their needs. Another possible enhancement would be collecting more advanced data points, that are not available within the OBD port are locked down by the manufacturer's software. If an agreement is made with a car manufacturer, more data could be collected for the system. Another good enhancement would be an automatic power device. This would be a device that turns on the Raspberry Pi automatically to start collecting data when the car is started, rather than having to push a button. Beyond these, all the system components could be improved and more efficient if more money was available. Instead of using a full Raspberry Pi, a dedicated microcontroller with a Wi-Fi module could do the job without all the overhead of the operating system.

## 4. Test Report

| Test ID | Requirement/ Constraint | Test Description | Expected Outcome | Actual Outcome | Pass/Fail |
|---|---|---|---|---|---|
| TC001 | Collect data from OBD II port | Verify the system's capability to collect data from the car's OBD II port continuously. | Data from all available OBD II parameters is collected without interruption. | Data was successfully collected without interruption. | Pass |
| TC002 | Save collected data in a local database | Confirm that the system saves all collected data to the local database accurately. | All transmitted data is accurately saved in the database. | All data is accurately saved in the local database. | Pass |
| TC003 | User interface for live data | Test the responsiveness and accuracy of the live data display on the touchscreen interface. | The interface responds within 250 ms and displays accurate data. | Response time within 200 ms; data displayed accurately. | Pass |
| TC004 | Display engine codes with severity | Evaluate the system's ability to display engine trouble codes and assess their severity. | The system accurately displays trouble codes and provides a correct severity assessment for each. | Trouble codes and severity are accurately displayed. | Pass |
| TC005 | Upload data to a web server | Test the system's ability to upload collected data to the specified web server. | Data is successfully uploaded and matches the local database entries. | Data was uploaded successfully and verified on the server. | Pass |
| TC006 | Web server data collection and storage | Ensure the web server correctly collects and stores received data. | Web server receives all data packets and correctly stores them in the database. | Data is correctly received and stored by the web server. | Pass |

| | | | | | |
|---|---|---|---|---|---|
| TC007 | Efficient database operation | Verify the database's efficiency in data storage and retrieval processes. | Operations completed within reasonable time frames and without excessive resource consumption. | Database operations are efficient and within expected resource usage. | Pass |
| TC008 | Web application functionality | Assess the web application for viewing, comparing, and analyzing collected data. | All features function as intended, with data accurately represented. | Web application functions correctly; data representation is accurate. | Pass |
| TC009 | Size and Weight | Measure the final assembled system to confirm it fits within the volume constraints. | Dimensions do not exceed 25x20x10 cm, and the system is lightweight. | Measurements confirm compliance with size and weight constraints. | Pass |
| TC010 | Power Consumption and Supply | Monitor the system's power consumption to ensure it does not exceed 15 Watts. | Total power consumption is under 15 Watts. | Power consumption was measured at 14 Watts. | Pass |
| TC011 | User Interface Response Time | Evaluate the latency of the user interface under various conditions. | All user actions receive a response within 250 ms. | All actions responded to within 250 ms. | Pass |
| TC012 | Data Transmission and Connectivity | Test the Bluetooth connectivity range and data transmission latency. | Stable connection with latency under 500 ms at all tested distances. | Bluetooth connection stable; latency at 400 ms. | Pass |
| TC013 | Durability and Vibration Resistance | Test the system's resistance to vibrations and ensure structural integrity. | No malfunction or loss of structural integrity was observed. | Withstood typical vehicle vibrations without issues. | Pass |

| | Test Case ID | TC001 | TC002 | TC003 | TC004 | TC005 | TC006 | TC007 | TC008 | TC009 | TC010 | TC011 | TC012 | TC013 | Total Test Cases |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Requirement ID | | | | | | | | | | | | | | | |
| L1.1 | | X | X | | | | | | | | | | | | 2 |
| L1.2 | | X | X | | | | | | | | | | | | 2 |
| L1.3 | | | X | X | X | | | | | | | | | | 3 |
| L1.4 | | | | X | X | | | | | | | | | | 2 |
| L1.5 | | | | | | X | X | | | | | | | | 2 |
| L1.6 | | | | | | X | X | X | | | | | | | 3 |
| L1.7 | | | | | X | X | | X | | | | | | | 3 |
| L.1.8 | | | | | | | | X | X | | | | | | 2 |
| Constraint ID | | | | | | | | | | | | | | | |
| SC1 | | | | | | | | | | X | | | | | 1 |
| SC2 | | | | | | | | | | | X | | | | 1 |
| SC3 | | | | X | | | | | | | | X | | X | 3 |
| SC4 | | | | | | | | | | | | X | X | | 2 |
| SC6 | | | | | | | | | | | | X | | X | 2 |

## 5. Appendices

- Software Source Code: https://github.com/SamNicklez/senior-design-final-project
- PythonOBD Library: https://python-obd.readthedocs.io/en/latest/
- Kivy Documentation: https://kivy.org/doc/stable/guide/basic.html
- KivyMD Documentation: https://kivymd.readthedocs.io/en/latest/
- Emulator: https://github.com/Ircama/ELM327-emulator