Drivaid: A Smart Driving Monitor

Authors: Samraj Kalkat, Reid Yesson, Ryan Vimba: Electrical and Computer Engineering, Carnegie Mellon University

Abstract—**Drivaid is a system that is designed to alert drivers of infractions and bad driving habits in real time and log these on a user-friendly web application on their phone. The hardware connects to the user's car's internal computer and makes judgments based on the driver's real time habits. Drivaid can be used to improve and monitor driving for personal and professional levels.**

Index Terms—**Adafruit GPS, MQTT, Raspberry Pi, PiCAN2, OBD-II, ELM327, JSON**

1 INTRODUCTION

Drive monitoring has been a secret of the automotive and insurance industries for years, with little room for consumer use. Why not use that information to make us better drivers? The way we have learned to drive has not changed over the past 50 years - an instructor sits in the opposite seat and controls the students' gas and brakes. Drivaid is looking to bring that diagnostic information directly to the user while driving. Our product will better inform drivers of their mistakes and log their errors to learn from them. Drivaid can also be used by corporations that rely on ground transportation or use company cars; Drivaid can provide driving scores to ensure company equipment is used safely. Drivaid will be stored in a custom enclosure to guarantee that equipment is not affected by car movement and not impair the driver's line of sight or overall comfort while driving.

In terms of design, infraction messages will be communicated from our Raspberry Pi to the user's smartphone in a holder on the dashboard. Our Raspberry Pi 4 connects to an ELM327 cable, which reads data from the Onboard diagnostics (OBD-II) port every second and detects infractions in real-time. The ELM327 contains a programmed microcontroller that presents a command protocol to interface with and translate the messages from the OBD-II port. Our program pings the car's diagnostic information via the OBD-II Parameter IDs (PIDs) every second to determine infractions. Infractions include:

- Speeding up or slowing down too quickly.
- Revving the engine too high.
- Driving above the speed limit.
- Driving while diagnostic trouble codes (DTC's) are illuminated.

The analysis programs will then send data and infraction messages via MQTT to the web application on a smartphone.

2 DESIGN REQUIREMENTS

To ensure that the Drivaid system correctly serves its purpose as a helpful point of reference for bad driving habits and driving infractions, we have outlined several design requirements which will ensure that the system is effective.

Our first set of requirements relates to infraction detection and exactly what needs to be detected from the data to make the system effective. They are as follows:

- Detect if the user drives too fast in a given speed limit, drives too slow in a given speed limit
- Determine if the user accelerates too fast at any given point of time
- Measure and display fuel consumption
- Calculate driving efficiency based on acceleration and RPM patterns

The next set of requirements relates to the functionality of the system, such that it is a smooth, understandable, and easy-to-use process for the user:

- Must be able to accurately return the requested PIDs and return data without corruption to 99 percent accuracy
- Data must be accurate and mirror real-world condition of the actual readings from the vehicle (Ex: RPM reading of 1800 is actual RPM of the car)
- Alerts must display on the web app within 3 seconds of real-time occurrence
- Collected data from a trip must be displayed in easy to read / graphical form on the website that the user can quickly understand
- Driver must be made aware of the infractions made during their trip
- Compact and non-intrusive closure of electronic components in the vehicle that will stay mounted to the dashboard and not obstruct the vision or movement of the driver

By following these requirements, we will ensure that our system is accurate and achieves its goal of showing a driver relevant data about their driving habits to understand and improve their driving. To test data accuracy, we will test each point of data that we plan to use from the OB-II logger and track these values to ensure they are accurate with what we are doing in the vehicle, which is the essential requirement. Without knowing that the data we are receiving is correct, our infraction checking and web visualization will appear nonsensical to the user. Since most of these values also have a reasonable bound of how low or high they might be, we can add run-time checks to make sure we are not reading or producing any abnormal values.

3 ARCHITECTURE OVERVIEW

The purpose of our system is to collect data from a vehicle while a user is driving, analyze this data, and provide real time feedback to the driver as well as a log of the users driving analysis. To provide these features, our system architecture can be broken down into three main subsystems. These subsystems and overall system architecture can be seen in our block diagram in the Appendix.

The first subsystem is the hardware connection between the RPi and the OBD-II port on the vehicle. We connect the RPi to the OBD via an ELM327 cable. The ELM327 cable is a programmed microntroller which allows us to request and receive CAN data from the OBD port. This connects to the RPi via USB. We can then run a Python script on the RPI to send PID requests to the On-board computer of the vehicle every second through the ELM cable and receive back an un-formatted 29-bit OBD response. Once we receive this data, we store it into an SQLite database on the Raspberry Pi for ease of use in calculating infractions and sending the data to the website.

Another hardware component is the Adafruit GPS module with antenna. This is also connected to the RPi via USB and is what we use to collect the Latitude/Longitude position of the vehicle every second. This information is also stored into the database and is passed later on to an API which we use to determine the speed limit.

We power all of the hardware through the 12V cigarette lighter plug on the car. We use a UBEC DC/DC step-down converter to convert this to 5V since this is a safe voltage to power the RPi.

The next subsystem is our data analytics software program. Once the data has been collected into the SQLite database on the RaspberryPi, we have another Python script to interpret this data and determine if any of our specified driving infractions are committed. We check the speed limit, fuel efficiency, RPM efficiency, motion efficiency, and the diagnostic trouble codes of the vehicles. The details and equations of these checks are specified later in the system description. We use threading in our program to read all the data and compute all of the different infractions simultaneously.

The final subsystem is the website and user interface. Once the data has been collected into the Database and the infractions have been calculated through our algorithms, we send the data to an online database to display to the user. We send the data using an MQTT protocol. Since our data is mainly numerical, MQTT offers the most lightweight and efficient solution to send the data. The data is sent via MQTT to ThingSpeak which is an online tool we use to temporarily store the data to display infractions.

If an infraction is detected and sent to our website, we trigger a clearly labeled light indicator on the website within 1 second, showing the user that they have violated an infraction. At the same time, the raw data from their drive is being stored on the website and formatted into graphs for them to analyze once they are done driving. This raw data is also stored onto an online PostreSQL database so that it can be viewed and referenced later.

In terms of modifications from our original design, there were a few changes we made to accommodate issues and unexpected behavior we were experiencing. One major change from our original design was our hardware interface with the OBD-II port. Originally, we planned to use a PiCAN2 which is a CAN Bus attatchment for the Raspberry Pi to interface with the OB-II, however we had difficulties with compatibility with our test vehicles and were not able to read data. This is why we switched to the ELM327, which allows us to read the same data just as conveniently.

Another change we made was which infractions we checked for in our algorithms. We had originally planned to also check the turning angle and radius of the car as well as the breaks, but we learned that this data was not easily accessible with the standard OBD-II PIDs so we had to remove it from the checks.

Other than these changes, the rest of the system is similar to our original design plans and works as we had expected.

4 DESIGN TRADE STUDIES

4.1 Speed Limit Calculations

We had two potential API options to gather speed limit data, the Roads API and the OpenStreetMaps API. The main difference between the two is that Google maintains the Roads API, while OpenStreetMaps is community-owned and run by the OpenStreetMaps Foundation. The Roads API has a much larger abundance of data because Google has put many resources into gathering data across the country. Because they have some of the best data in the mapping department, these APIs are not free to use. I go into much greater detail in the Appendix about the exact costs of these API calls and our strategies for minimizing these costs. Still, in the end, we could not maintain low prices for everyday driving while ensuring our other crucial requirement of real-time speed limit infraction notifications.

Additionally, even if we could find a good balance between latency and cost, we ran into licensing issues with Google. When we tried using their speed limit API calls, we realized that this specific API required explicit approval only for large projects that plan to use the API in at least 500 vehicles. Because our project fell well short of that, we decided to go with our second option, the OpenStreetMaps API.

The OpenStreetMaps API had no license requirement since it is community-driven, and it allowed us to eliminate the day-to-day API costs of using our product. However, using a free API came with its challenges. Firstly, the data for Pittsburgh is incomplete. Sometimes we would make an API call with our vehicle's GPS coordinates and receive a response with no speed limit because this street had no speed limit data. To compensate for this, we decided to limit our testing zone to areas such as Squirrel Hill, Oakland, and Shadyside. These areas only lack data on smaller side streets, which we know have speed limits of 25 miles per hour. So when we did not receive any speed limit for an area while driving in these zones, we defaulted the speed limit to 25.

Secondly, the OpenStreetMaps API response times had a significant variance. Some of them would arrive in less than two seconds, but others would take up to 20 seconds. This variance was an issue because all our analysis functions ran sequentially. If the speed limit function took 20 seconds to complete, the other functions would also be behind by 20 seconds. This delay is a problem for maintaining real-time notifications of all our analysis functions. We moved the speed limit analysis function to a separate thread to remove the cascading delays caused by the speed limit API calls.

Additionally, because we collect data from the onboard computer every second, we could not evaluate every vehicle speed that our data collection process gathered. Instead, we consistently assessed the most recent speed in the database. Even though we can't assess all speeds gathered, the ones we evaluate reflect the vehicle's current speed.

4.2 ELM-327 Cable

There are many devices on the market that are OBD-II to Bluetooth, and one of our original designs included a Bluetooth connection directly to the phone instead of using a Raspberry Pi. The phone would in turn use its own GPS to measure speeding violations. This solution would have been clean with no extra hardware or charging necessary, and everything could have been done in the web app backend. However, we wanted to incorporate a hardware angle to our project in the spirit of getting exposure to more areas of Electrical and Computer Engineering. In addition, we are able to do more data analytics on the Raspberry Pi end with 64 empty GBs on the SD card and 8GB of RAM. The split between computing on the Raspberry Pi end and user experience on the web application makes working in tandem much easier and eases the computing load to a more capable device.

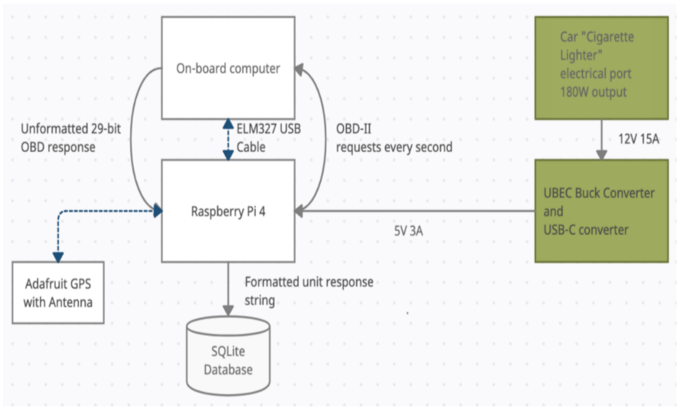
4.3 Mobile Hotspot

Another aspect of our design in which we considered many trade offs was how the web application should connect to the Internet and how the Raspberry Pi should send messages via MQTT. We considered buying a mobile data module for the Raspberry Pi, but thought satellite-based connection would increase our latency and affect our real-time alerts. We therefore settled on using the phone that would be on the dash holder also act as a mobile hotspot, which allows the Raspberry Pi to communicate with the Web Application without external processes slowing the system down.

5 SYSTEM DESCRIPTION

As seen in our architecture overview, our system consists of three main subsystems that will come together to produce our final product. The first is the OBD-II data logging system, which will handle communicating with the on-board computer of the vehicle, gathering data, and formatting/storing the data. The second is the data analysis. This the bulk of our system where we will be developing algorithms to calculate infractions and warnings based off of the collected data. The outputs of these algorithms will then be sent to an online database for storage. The final subsystem is the web-interface where we will be taking all the data that has been collected and analyzed, and display it in a convenient and real-time interface for the user to see.

5.1 OBD-II Data Logging



On-board computer
Car "Cigarette Lighter" electrical port 180W output
Unformatted 29-bit OBD response
ELM327 USB Cable
OBD-II requests every second
12V 15A
Raspberry Pi 4
UBEC Buck Converter and USB-C converter
5V 3A
Adafruit GPS with Antenna
Formatted unit response string
SQLite Database

Figure 1: OBD-II Block diagram

This subsection has to do with the electronics and systems that are closest to the on-board computer on the car. First, we have a ELM327 cable that goes from the car to the PiCan connected to the Raspberry Pi. We can run a program using the CAN library in Python to ping the computer for data every second. The PiCan simplifies the firmware between the Raspberry Pi and the on-board computer and allows us to have TX and RX programs that can run in parallel. Our TX program sends out a request string to the CAN that requests certain fields. Afterwards a message in CAN format will be received by our RX program, and our program will be responsible for separating them into specific known data fields (called PIDs). Standard PIDs such as velocity (mph), engine speed (RPM), DTC (diagnostic trouble codes), and fuel level can be knowingly found from the message. Other PIDs such as steering wheel angle and turn signals are non-standard and vary from make to make and require analysis of the message to be detected. Many PIDs are blocked from the OBD-II port and are not dropped from the on-board computer onto a readable data line, such as odometer data and steering wheel info in our case. More discussion of standard and non-standard PIDs can be found in the references [3]. After indexing into the CAN message we receive the following information to store in the SQLite database:

- Fuel level
- Velocity
- RPM
- Throttle position
- DTC: oil level
- DTC: tire pressure

After receiving the data, we store these six data points in a SQLite database. For the DTC error codes, we are using a binary classification: 0 if it is off, 1 if it is on. For throttle position, we get a number between 13.3% and 80% - this number is normalized later in our analysis phase to get a better sense of motion efficiency. Velocity is also multiplied by the kph to mph factor, .6214. We also add a seventh data point: acceleration. The acceleration in meters per second can be calculated by:

$$1609.34 * (v_c - v_{c-1})/1s$$

where v_c is the current velocity and v_{c-1} is the velocity 1 second beforehand. This difference is then multiplied by the mile-to-meter conversion multiplier, 1609.34. Acceleration is used to determine the motion efficiency of the car - if the user brakes or accelerates too fast.

After the data is parsed, the data is fed into the SQLite database, which is in-memory and stored on a 64GB SD Card on the Raspberry Pi. This was one of our tradeoffs: using an in-memory database which takes more space instead of sending out data to the web once we got it and storing it on the web server. We thought an in-memory database made the most sense because our data size is relatively small: a tuple made of seven data fields and a timestamp. This tuple is also broken into analysis data and user graph data to be sent to the website in separate HTTP POST requests. We can then send straightforward time-based queries to the SQLite database in real-time to make logs to send to the web server. Here is an example of a tuple from our database with units added (separated across two lines):

Timestamp	Fuel Level	Throttle	RPM
5/10/21 22:07:15	3gal	13.5	200rpm
DTC Oil	DTC Tires	Velocity	Accel.
0	0	41mph	.45m/s^2

This data will then be correlated with the GPS data using a foreign key, the timestamp.

The Raspberry Pi electronics are powered by the car's cigarette lighter plug, which outputs 12V and 15A of current. In order to power the Raspberry Pi, we need 5V with much less current draw to use the USB-C charging port. Therefore we are using a UBEC Buck Converter to power the device, which alleviates problems with dying batteries. Any more than 5V on the Raspberry Pi end is potentially harmful, so we conducted thorough tests with a multimeter to measure output current voltage.

To retrieve the location of the driver, we use an Adafruit GPS which is connected to the Raspbery Pi via USB. The GPS has an antenna which is attatched to the top of the car with a magnet. The GPS gets a fix of the car's posistion every second and this result is captured and stored int eh SQLite database.

5.2 Data Analysis

To analyze a driver's performance, the program uses information from three sources.

1. Raspberry Pi local SQLite database
2. Adafruit GPS
3. OpenStreetMaps API

The analysis subsystem of Drivaid analyzes data from the SQLite database and the OpenStreetMaps API responses. It comprises 5 analysis functions and runs on three threads, one thread to control the GPS, another to run the speed limit analysis function, and a third to run all the other analysis functions.

For the entirety of a user's drive, the two analysis threads continuously check the SQLite database for new entries in the data tables. If they find new data, the data is passed along to the proper helper function to check whether or not that data provides evidence of a driving infraction. The function results are then placed into fields in a URL and sent to the web application.

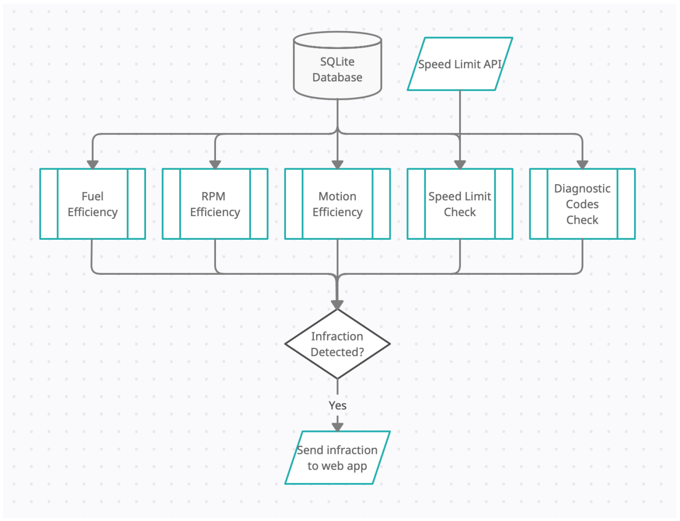


Figure 2: Analysis Overview

Speed Limit: To check for a speed limit violation, we compare our speed with the speed limit of our location. If our current speed is s_c and our current speed limit is s_l and $s_c > s_l$, then the function output reflects a speed limit violation.

However, because there is a wide variance in the time the GPS needs to get a coordinate fix from a satellite, we do not wait to be given a speed before we request a location from the GPS. If we tried to get our location from the GPS only after we have a vehicle speed, the location we receive could be hundreds of feet from the place our speed was measured. If the speed limit changes in this distance, we could produce an incorrect infraction.

In order to ensure we get the closest GPS coordinates to the location the velocity was measured, we continuously receive GPS coordinates along our drive and log them in a separate table with the timestamps they were measured. This table is helpful because it allows us to access the latitude and longitude received closest to the time the velocity was received. We do not have to wait for another set of coordinates to come in from the GPS. This quick lookup time is why it is essential to run the GPS in a separate thread.

Fuel Efficiency: To measure the fuel efficiency of a user's drive, we compare the distance the vehicle traveled with the fuel level to measure how much fuel the vehicle uses per unit distance. We were planning on using the odometer data tracked by the on-board computer; however, that information is nonstandard, so we had to come up with a way of measuring distance. We did this by using the current speed, our previous speed, and their respective timestamps. We estimate distance by multiplying the average of the two speeds with the time change in the time between the two timestamps. This distance represents the distance traveled between the previous and current measurements. We add this to our previous total to get our new total distance traveled, d_{total}. To get the total fuel used, we subtract the first fuel measurement f_0 from the current reading f_n. We get fuel efficiency by $\frac{f_0 - f_n}{d_{total}}$.

RPM efficiency: To measure RPM efficiency, we check whether the engine is operating at maximum torque. Most modern engines operate at an efficient torque within a wide range of RPMs. Because maximum torque occurs within a wide range, this function only outputs an infraction if the driver is driving at an extremely high RPM (above 4500).

Motion Efficiency: Excellent acceleration and braking efficiency are difficult for a driver to achieve because the most comfortable acceleration is 0. We decided to measure whether a user accelerates at a rate that would be uncomfortable to a passenger. These are the rates that would affect the customer experience and cause doubt about a passenger's safety. If the function measures acceleration above 1.5 m/s^2 or below -1.5 m/s^2, the output reflects this infraction.

Diagnostic Trouble Codes: The diagnostic trouble codes (DTCs) all arrive in one response from the CANbus. We check all the responses' values, comparing them with the codes associated with oil level and tire pressure. If we receive codes that match either of these known values, we return this information from our function.

Data Transmission: Once we have completed the analyses as defined above, we need to send these outputs to the web application using Thinkspeak. ThingSpeak is an open-source Internet of Things application and API to store and retrieve data from things using the HTTP and MQTT protocol over the internet. We create a URL that will go to our web application and place each of the analysis function outputs in specific fields that the web application knows how to interpret.

Our RaspberryPi connects to the internet via a mobile hotspot. We assume the driver will have a smartphone capable of creating a hotspot network from their carrier data. The RaspberryPi will be connected to this network to send the data to the online database.

5.3 Web-Interface

This final subsection explains how we will be receiving the analyzed and raw data from the logger and algorithm output and displaying in a simple report and notification interface for the driver to view online.

For our back-end system, we hosted our website on GitHub Pages since this provides a free and fast solution for us to host our page. We use MQTT via a Python script to send our data from the RaspberryPi scripts to the ThingSpeak API. ThingsSpeak is used to receive the MQTT data which we then format and display on our website. This data is then stored into a PostgreSQL database to be referenced later.

For the front-end of our website we use basic HTML and Javascript to create the overall layout. The website has two main pages, the infraction notification page and the data summary page. The infraction notification page can be seen below. It displays several lights indicating each of our infractions and when an infraction is detected and a notification is sent to our web server, the light turns on. The lights change within 1 second of the infraction being detected in our algorithm. This allows for the user to know exactly when they commit an infraction. We also display the current speed limit of the road that the user is driving on which is found in the OpenStreetMaps API and sent to the website in the MQTT data stream. We designed the interface like this so it is easy for a driver to see while they are driving without it being intrusive to their vision or overly distracting.

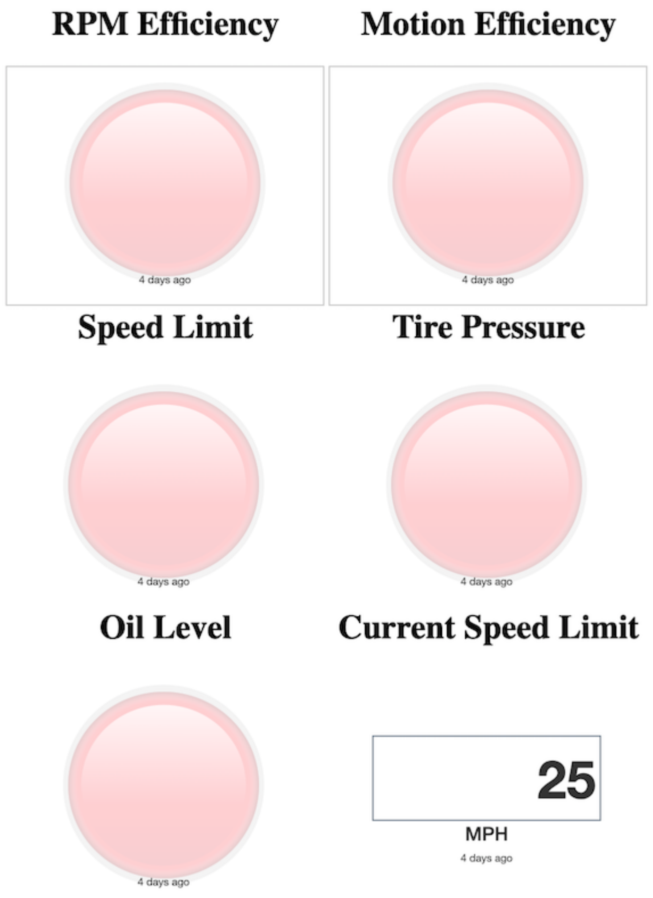


Figure 3: Infraction Notification Web-Page

The data summary page displays all of the raw collected data from the OBD-II readings in a graphical form, as seen below. We display graphs of the fuel consumption, throttle, speed, RPM, and acceleration over time for the user to visualize their data throughout their drive. The graphs are interactive and allow the user to zoom in on specific points of data.

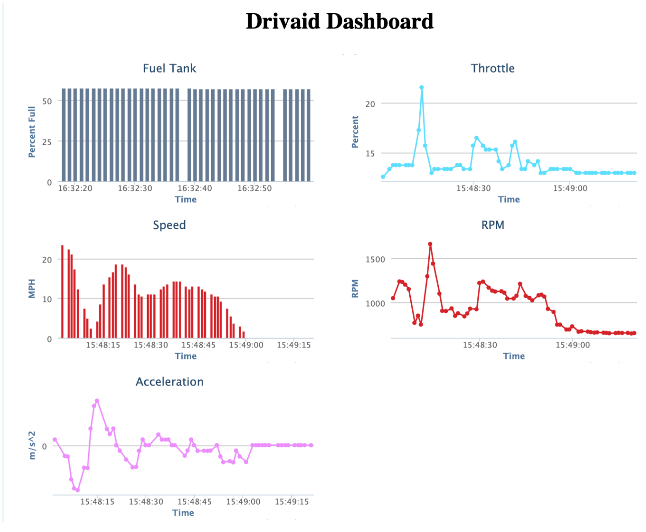


Figure 4: Data-Summary Web-Page

A simple visualization of the entire web system can be seen in the figure below.

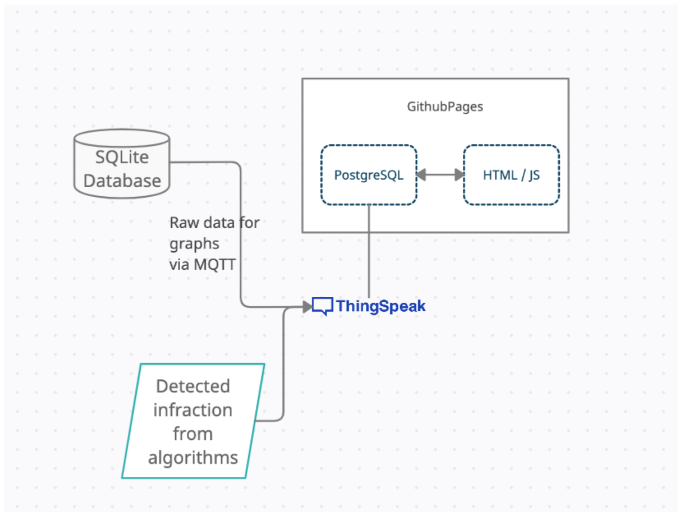

GithubPages
SQLite Database
PostgreSQL
HTML / JS
Raw data for graphs via MQTT
ThingSpeak
Detected infraction from algorithms

Figure 5: Website System

6 TEST & VALIDATION

In order to ensure that our requirements are met and our overall system is functioning correctly, we implemented several tests on each of our subsystems and compared them to our desired metrics. These tests prove that our system is functioning correctly and each of the system requirements is being consistently fulfilled. Each of our testing metrics and validation techniques are explained in detail in the following sections.

6.1 Results for API Request/Response Time Testing

One of our requirements from the beginning of the project is that we wanted to receive OBD-II data from the car every second and have the website refresh every second - this was our definition of "real-time". Any more than that, the user would not have any clear idea of their mistakes and delays would be confusing and detract from the system's usefulness. After switching to the OpenStreetMap API and using two API calls for better locational accuracy, we realized during testing that the API calls were taking too long and hanging the rest of our system. One test we did during a drive over thirty seconds yielded 13 points with high variance - some took 4.5 seconds. These results are shown below:

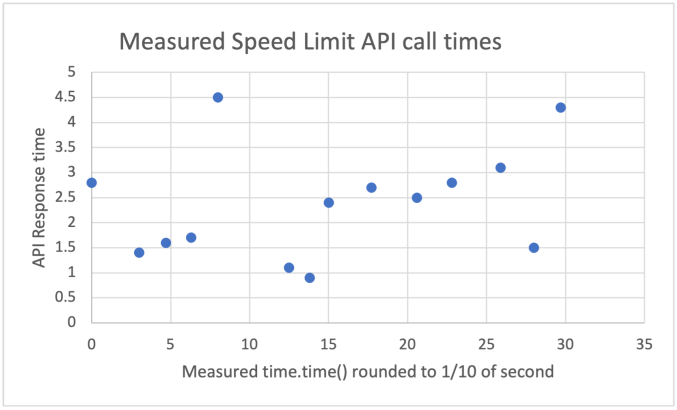


Figure 6: Measured driving test of API times

Some other testing showed that we would only receive an API response after 15 to 20 seconds at the max. Before our solution, this would hold up our OBD-II logging, and after the API call finished, it would read velocities, RPM, throttle, etc. from 15 to 20 seconds ago. We could isolate the problem to the API because we were testing in a populated area and we were clearly receiving the GPS coordinates. We could not improve the API response time, which meant that we would need to separate this process from the rest of the code. After seeing and measuring these results, we placed the API calls in their own thread alongside the three other threads for GPS reads, OBD-II writes, and OBD-II reads. We did not account for this delay in our initial specification and though much of our delay would come from requesting the data from the car.

6.2 Results for OBD-II Data Logger Testing

In order to test the connection to the OBD-II port and the data we receive from our PID requests, we implemented several tests to validate each of the requirements of this system.

The first requirement we tested is that we can request and receive the desired data from OBD-II. To test the accuracy of this data, we setup a live stream of our data in a terminal and inspected the data as it came in through our script. We then compared this data to the data that is displayed on the dashboard of the vehicle to ensure that it is correct and accurate. We also carried out different tests by revving the engine, accelerating and decelerating, etc. and observing if our system is updating accurately which it did.

The next requirement we tested is that our data is well typed and correlates with our SQLite database format. As we receive data through our loggin script, we implement checks in Python to make sure that the data being stored in each row of the database is in the correct format and is of the correct type. If it is not, we disregard that datapoint and continue the script. This ensures that we do not have any malformed data in our database or MQTT request strings.

6.3 Infraction Detection Algorithm Testing

To test our algorithms, we simulated driving data and analyzed the results to ensure they catch the infractions they are supposed to detect. For the Speed limit analysis function, we tested locations with different speed limits in Pittsburgh like Forbes Avenue, Fifth Avenue, and the highway, which have speed limits of 25 MPH, 35 MPH, and 55 MPH, respectively. We tested different speeds in each location and verified that the different zones compared the correct speed limits against the vehicle speeds from the vehicle. Additionally, we did a couple of tests driving from a 25 MPH zone to a 55 MPH zone and a couple of tests driving from a 35 MPH zone to a 25 MPH zone. Within 10 seconds of switching roads, we received updated speed limits.

For the fuel efficiency test, we did a 10-minute drive and compared the fuel efficiency measured by our system against the fuel efficiency displayed by the car. The result was within 5 miles of the actual fuel efficiency.

Our test for the RPM detection and motion efficiency was similar to the speed limit check - we tested data points above and below the safe RPM and acceleration levels - and the system detected all the unsafe data points. Specifically, we tested 100 data points with RPM levels above the level we allow, and our RPM infraction function detected exactly 100 infractions. Similarly, we tested 50 intervals with acceleration above $1.5m/s^2$ and 50 intervals that had acceleration below $-1.5m/s^2$. Exactly 100 points were detected, validating our motion infraction function.

6.4 Web Application Testing

Our first test for the website was to ensure data sent from the raspberry pi arrives at the web application within 5 seconds. We tested this by sending data and observing the response time, verifying that the website consistently updates about 5 seconds after the data is sent. Our second test was for data accuracy. We created a test script that sent various data displayed in each of our different sections on the website. We found that the data sent was consistently correct and displayed in the correct places.

6.5 Integration Testing

After we tested and corrected the individual subsystems of our project, we tested our system integration by setting up the whole system in the car and making sure data would pass data through each of the systems correctly. Our first test was to drive very inefficiently with lots of quick acceleration and braking. Our system displayed very low efficiency scores, which we see as a success. We also tested by revving the engine extremely high, which resulted in the RPM infraction light going off on the web application. We realized we had to tweak a few of our analysis functions through this testing because they were too sensitive to changes in acceleration and throttle. We noticed that even if a person is keeping their foot in the same position on the pedal, the readings from the CANbus can vary slightly, which was causing some unexpected infractions. To deal with this, we incorporated some slack in the analysis functions.

7 PROJECT MANAGEMENT

7.1 Schedule

Our full schedule can be found in Appendix B at the end of the report. We have separated the tasks into the main subsystems of our project and broke each of those systems down into pieces which work in conjunction with the other tasks that are happening in parallel.

7.2 Team Member Responsibilities

Samraj Kalkat

- Designing web application to display real time infraction and driver logs
- Setting up MQTT data stream to send data from algorithms and OBD data logger to the website
- Helping with OBD-II data request and interface scripts

Ryan Vimba

- Design local SQLite database for vehicle and GPS data
- Design algorithms to analyze the data and output infractions
- Sending analysis outputs to web application

Reid Yesson

- Connecting hardware components to establish connection with OBD-II port
- Writing scripts to request specific data from the CANBUS
- Formatting data and storing in mySQL Database
- Helping with OpenStreetMap API setup and response

7.3 Budget

In our design report, we did not account for the parts that would be necessary to link the system to our laptop for testing. These parts are the USB-C to Ethernet, the 5 foot Ethernet cable, and Micro HDMI to HDMI cable to connect a monitor.

Every part was bought with our budget. The Raspberry Pi, PiCan2, and VGA to OBD-II cable provide the computing and data collection aspect of our project. The Cigarette Lighter converter, UBEC Buck Converter, and USB-C Breakout allow us to power the device from the car.

Part	Cost	Status
PiCan2	\$51.95	Received
Adafruit GPS	\$50.38	Received
ELM327 Cable	\$18.85	Received
Raspberry Pi	\$55	Received
UBEC Converter	\$9.95	Received
USB-C to Ethernet	\$16.60	Received
Ethernet cable	\$13.70	Received
Cigarette lighter breakout	\$9.74	Received
SD Card	\$12.99	Received
Micro HDMI to HDMI	\$5.98	Received
USB-C breakout	\$10.33	Received
Total	\$268.20	

7.4 Risk Management

For this project, most of our risk stems from testing our project. We can test the Drivaid with dummy data for GPS, but we will need to do road tests to make sure our project works in the real world. Dummy data is useful when determining speeding, but hard braking, hard turning, and fuel efficiency will all require some real-world counterpart testing. Our driving will need to be safe even though we will need to recreate unsafe infraction events, so we will need a wide open area to test the Drivaid.

Another risk comes with charging the Raspberry Pi from the car batteries via the cigarette lighter port. The output from the batteries is 180W, 12V 15A. After this voltage and current is passed through the buck converter, there should be 5V and 3A, which should power the Pi. Thorough testing must be done first on the UBEC before charging the Pi to mitigate the risk of passing too much voltage or current and frying the Pi and/or the PiCan, our two most expensive elements.

There is also a possible issue of losing signal during a drive under a bridge or a tunnel. In this case, it means that the Raspberry Pi will not be able to send the data via MQTT to the database and we will not have notifications and in theory lose some the data that is being read. This being considered, we implemented an exponential backoff to slow down the rate of data transmission until a connection is made. In the case that a connection is never established, we could also implement a procedure to send any unsent data on launch of the system. Doing this will make sure that if we are starting a new ride, all the data that was being held from the ride before is sent, so in theory it won't be lost to the user.

8 ETHICAL ISSUES

One possible failure is due to the use of geolocation location with the GPS that Drivaid uses to plot routes. If that information were to leak or end in the wrong hands, information about drivers' location can be at risk. The location data is also timestamped, which means that the exact location at any given time of a user could be found given knowledge of their identity. On Drivaid's end, this information needs to be encrypted properly so that hackers cannot find the location of drivers, especially student drivers who are likely to be minors. We are not implementing cyber security for this iteration of our design, but it would need to be added to protect the physical security of our user base.

Drivaid may also be susceptible to misapplication on the company end because it is entirely up to the company how they want to use the data in the case of bad driving. Right now, there is a trust system that is set up between drivers and their employers - if the vehicle was returned without any damage at the end of a shift, the employer can assume they drove safely. But with Drivaid, there is now infraction data and driving scores that the employer can track and use however they want. Employers can be unnecessarily harsh if they see bad driving from Drivaid after a shift, but that decision making is left entirely to their discretion, and Drivaid exists only to let them make informed decisions. The same could be said for any driving school or DMV that is looking at Drivaid data.

Drivaid uses a great amount of data from the car's OB-DII port along with geolocational data from our Adafruit GPS. The OBDII port has hundreds of data points that can be collected that describe engine status along with other pieces of information from the car's internals. Along with the GPS and time, Drivaid only uses the throttle, brake, vehicle speed, RPM, steering wheel angle, turn signal, and the fuel level. These seven data points are what we need to determine a number of infraction scenarios, and other information would not be useful to us and only serve as a burden to store in our database. Our data is a small subset of what we could be collecting from the car, but it also provides the most information about the driver, which is what we were aiming for. Another piece of information Drivaid does not collect is the identity of exactly who is driving the car since our device does not require it - everything is local to the car, not the user.

It was difficult to avoid these ethical issues during our design of the system since we did not build any cybersecurity requirements into our Minimum Viable Product not our final design. In future iterations, we would encrypt all of the data sent to the web application because of the sensitivity and the personal nature of our data.

9 RELATED WORK

One product that is similar to our device in terms of our logging and corporate monitoring are insurance devices that broadcast driving information. One of the more popular devices is Progressive's Snapshot device, which plugs into the OBD-II port and wirelessly transmits encrypted data to a third party.

After starting our project, we found a similar device made by Moto Safety that uses the OBD-II port to monitor and map infractions from a teen drivers' safety perspective. [2] The application is parent-only use and does not have real-time alerts. Their visualizations are similar to our proposed web application where the parent can view their teen driver's mistakes geographically and plots their trip on a map.

Another similar report to Drivaid is a report written by four undergraduate students at the University of Moturuwa in Sri Lanka. [1] This report was done for a similar undergraduate program with many of our requirements. Their design also includes real-time alerts that the driver can see. Unlike our design, this design is connected via Bluetooth to the OBD-II port but uses a much more complex web application with different user permissions that is designed with insurance companies in mind. There is therefore more cloud computing, which incurs higher AWS costs.

Our project is the first we could find that incorporates both real time alerts and Raspberry Pi computing, which we believe will result in faster alert times and less expensive costs in the long run. Since our system is real time, we wanted to cut down on the number of queries to remote servers and therefore believe our project is unique in the overall landscape.

10 SUMMARY

Our system met many of the design specifications, but throughout the project, we realized that interfacing with the car was much more difficult than we expected. For starters, the non-standard PIDs were difficult to capture, if not impossible due to the sensitivity of the data being returned by the car. We really would have liked to get steering wheel angle, and perhaps could have attached a gyroscope to the steering wheel, but that solution was had to commit to so late into the project and would not have done any favors for user safety and experience.

Our API calls were also different than we expected. We originally thought that we would be using the Google Maps Speed Limit API, but after realizing that we did not have the correct project credentials for Google approval, we needed to switch to OpenStreetMaps. We were generally happy with OpenStreetMaps, but still believe that Google Maps would have given more detailed information from one latitude/longitude pair, and probably would have been quicker with their response time.

10.1 Future Work

We are going to try to look more closely at the OBD-II codes from the Honda Civic and see if there is any more interesting information. We might not be able to do this with our ELM327 cable and may need a brand new PiCan2 board to be able to use the CANSniffer command, but I think it would be a neat solution. We were also looking at designing a custom PCB that includes our charging needs, data storage and MQTT protocol needs so we don't need to rely on the Raspberry Pi, which is large and takes up space in the car. It would also be a good way to get exposure to PCB design. We believe Drivaid could be an interesting product if it were more compact and hooked into the car directly from the port rather than with a wire. It will be difficult because the PIDs are not standard across different models and makes, but with a solution that encompasses the most popular models, it could definitely become a marketable product to companies with large private car fleets.

10.2 Lessons Learned

I think we learned that working with a defined system that is difficult to reverse engineer can be hard for a capstone project like this. There is a great amount of disagreeing data and outdated PIDs on the internet, which car manufacturers and insurance companies probably prefer. Car manufacturers are rightfully pretty stingy with the data they drop down onto their CANBus line, especially with recent car hacks and concerns about vehicle security as the computer is one of the most vulnerable parts in the car to attacks.

We also learned the difficulties of testing in a moving vehicle and all of the variance that comes with testing in real-time. It was much better to test with three people in the car so that the driver did not have to act as another researcher and could focus solely on commands from the one testing. When we testing with all three team members, tests went much smoother and we were able to isolate our issues much more easily than when we had a driver and one team member with a laptop in the passenger seat.

Glossary of Acronyms

Include an alphabetized list of acronyms if you have lots of these included in your document. Otherwise define the acronyms inline.

- MQTT – Message Queuing Telemetry Transport
- OBD – On-Board Diagnostics
- RPi – Raspberry Pi
- CAN - Controller Area Network
- PID - Process ID
- DTC - Diagnostic Trouble Codes

References

[1] De Zoysa Karunathilaka. “Driver Behavior Analysis using Vehicular Data”. In: *University of Moratuwa* (Feb. 2017), pp. 1 –57.

[2] *Moto Safety.* 2021. URL: `http://www.gpsmotosafety.com/index.html`.

[3] Wikipedia. *OBD-II PIDs.* 2021. URL: `https://en.wikipedia.org/wiki/OBD-II_PIDs`.

Appendix A

Our method to get the speed limit involves two API calls. The first API call is to $nearestRoad()$. $nearestRoad()$ takes as input up to 100 GPS coordinates and outputs a $PlaceId$ associated with each pair of input coordinates. A $PlaceId$ is a unique id from Google that identifies a specific section of road. The second API call is to $speedLimits()$ which takes as input up to 100 $PlaceId$'s and returns the speed limits associated with each of these $PlaceId$'s. The $nearestRoad()$ API costs \$0.01 per *query*, and the $speedLimits()$ API costs \$0.02 per *element*.

The trade-off is between the amount we are willing to spend and how accurate and responsive our speed limit analysis is. To ensure correctness and provide the most accurate speed limit data, we would call $nearestRoad()$ on every GPS coordinate as soon as we get it and follow it with a call to $nearestRoad()$ to get the associated speed limit. However, if we make both these queries once per second on a 60-minute drive, it would cost $(0.01 + 0.02) * 60 * 60 = \$108$, which is quite expensive for only one hour of driving. However, if we take advantage of the fact that $nearestRoad()$ is only \$0.01 per *query*, we can fit 100 pairs of GPS coordinates in each query, and can decrease our $nearestRoad()$ calls by 100x. This price-cutting method comes at the cost of responsiveness for the speed limit function.

We want to query the $nearestRoad()$ API as often as possible to make sure we compare our speed with the right speed limit. However, we don't want to query so often that we are wasting API calls if we have barely moved. We tentatively decided to make this query once per minute, costing \$3.60 per hour of driving.

Additionally, we can store a map between $PlaceId$'s and speed limits to cut down on calls to the $speedLimits()$ API. Because $speedLimits()$ is \$0.02 per *element*, it is a much more expensive call than $nearestRoad()$. If we cut down on $speedLimits()$ calls by storing the $PlaceId$ - speed limit association in our database, we only have to pay when we get a new $PlaceId$, not when we are on a road we have driven on before. Because people do the vast majority of driving in places they have already been, this cost will be meager once most speed limits are stored.

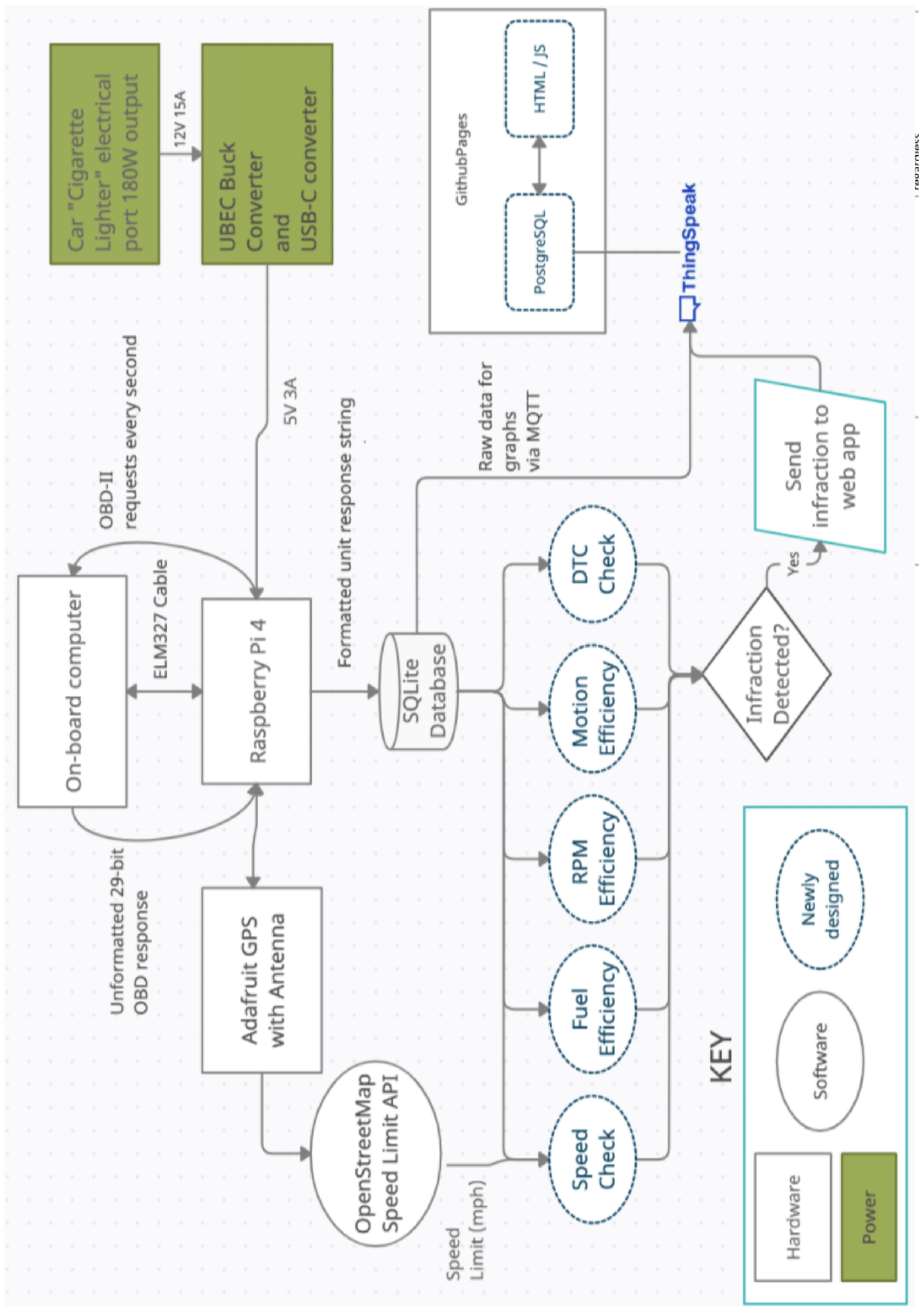

Car "Cigarette Lighter" electrical port 180W output
12V 15A
UBEC Buck Converter and USB-C converter
5V 3A
OBD-II requests every second
On-board computer
ELM327 Cable
Raspberry Pi 4
Unformatted 29-bit OBD response
Adafruit GPS with Antenna
OpenStreetMap Speed Limit API
Speed Limit (mph)
Formatted unit response string
SQLite Database
Raw data for graphs via MQTT
GithubPages
PostgreSQL
HTML / JS
ThingSpeak
Speed Check
Fuel Efficiency
RPM Efficiency
Motion Efficiency
DTC Check
Infraction Detected?
Yes
Send infraction to web app
KEY
Hardware
Power
Software
Newly designed

Figure 7: A full-page version of the system block diagram.

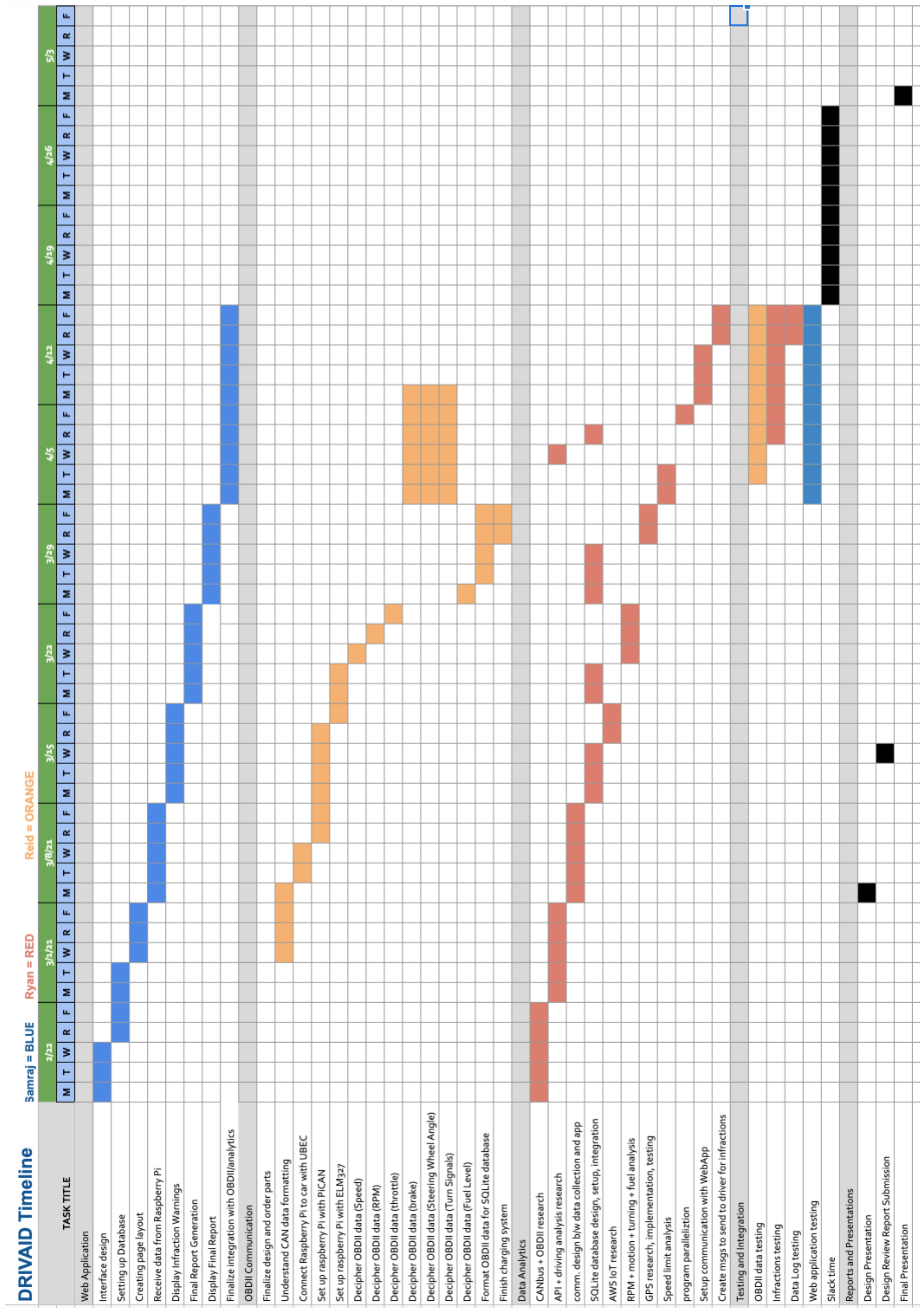


Figure 8: Gantt Chart