Pawel Jaglarz

20097569

BSc (Hons) in Software Systems Development

Final Year Project Report

20097569@mail.wit.ie

Table of Contents

Contents

Introduction

This project aims to develop a real-time OBD-II (On-Board Diagnostics II) data monitoring systems for most vehicles on the market. I would like to interface a live data monitoring screen into the vehicle so the user receives data on the go this data would later be used for example for preventative maintenance or find cause of an issue that arises with the vehicle. OBD – II uses in car sensors to receive data that may not naturally show up on the cars on board display unit. My project would allow users to more closely monitor those sensors such as engine RPM, coolant temperature, boost pressure etc… I decided to create this project because of my passion of cars and wanting to take care of my car better through using this device to save money time and prevent stress not knowing what is exactly wrong with my car.

Functional Summary

The main goal of this project is to provide a real-time OBD-II (On-Board Diagnostics II) data monitoring system that improves user vehicle management. Through the system's live data monitoring interface, customers can obtain necessary information on the performance and well-being of their car. The system's primary features include:

Real-Time Monitoring: This feature allows users to view real-time data from multiple in-car sensors, including boost pressure, coolant temperature, and engine RPM, providing quick insights into how well the vehicle is doing.

Preventative Maintenance: The system enables users to track vital metrics over time, allowing for early detection of potential issues. This proactive approach helps in scheduling maintenance before problems escalate, ultimately saving time and money.

Diagnostic Support: In the event of a vehicle issue, users can utilize the data collected to identify the root cause, streamlining the troubleshooting process and reducing reliance on mechanics.

User-Friendly Interface: The live data display is designed for easy navigation, ensuring that users can quickly access the information they need without distractions while driving.

Enhanced Vehicle Care: By creating a deeper understanding of the user's vehicle's systems, users can take better care of their cars, leading to improved longevity and performance.

Assumptions

Vehicle Compatibility: It is assumed that most vehicles on the market are equipped with the OBD-II port and sensors. The systems are expected to work as effectively as possible with most petrol and diesel vehicles manufactured after the year 1996

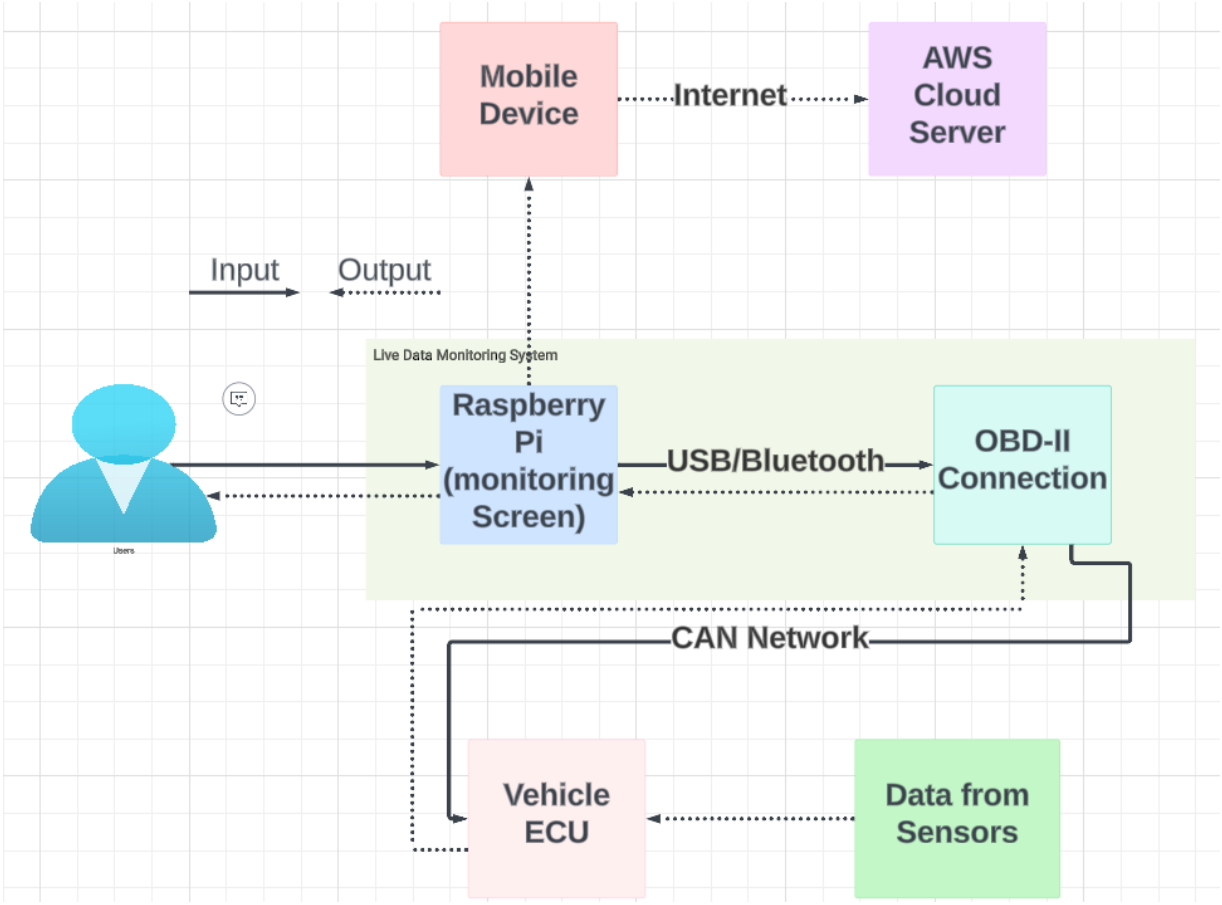
Users Technical Proficiency: Users are assumed to have basic understanding of their vehicles systems and technology. They should be comfortable the device to their vehicle and navigating the user interface. (Plenty of video tutorials on majority of cars and their OBD-II connection point location)

Data Accuracy: It is assumed that the data retrieved from the vehicle's OBD-II system is accurate and reliable. The system does not account for potential sensor malfunctions or discrepancies in data reporting.

User Environment: The system assumes that users will maintain a safe driving environment and not get distracted by the live screen instead focusing on the road.

Functional Requirements

System Context:



Components:

1. Users
    - Interacts with the system via Raspberry Pi interface.
2. Raspberry Pi (Monitoring Screen)
    - Central processing unit for live diagnostics.
3. OBD-II Connection
    - Connects to the vehicle's ECU to collect diagnostic data.
4. Vehicle ECU
    - Provides sensor data and diagnostic trouble codes (DTCs).
5. Mobile Device (Hotspot)

    - Used as a mobile hotspot to facilitate data uploads.
6. AWS Cloud Server
    - Stores historical data and allows for further processing and analysis.
7. Data from Sensors
    - Source of real-time diagnostic information.

Data Flow:

- User to Raspberry Pi:
    - Input/output interactions for accessing diagnostics.
- Raspberry Pi to OBD-II Connection:
    - USB/Bluetooth communication to retrieve data from the vehicle.
- Raspberry Pi to Vehicle ECU:
    - CAN Network for accessing diagnostic data.
- Raspberry Pi to Mobile Device:
    - Data uploads for cloud synchronization.
- Mobile Device to AWS Cloud Server:
    - Uses mobile hotspot to upload data to the cloud.
- Vehicle ECU to Data from Sensors:
    - Diagnostic data flow from sensors to ECU.

Requirements List

1. User Interface (UI)

1.1 The system shall provide a user-friendly interface for displaying live diagnostics. 1.2 The system shall allow users to input vehicle identification numbers (VIN). 1.3 The system shall display real-time data, including vehicle speed, RPM, and engine temperature. 1.4 The system shall provide graphical representations of data (e.g., graphs, gauges). 1.5 The system shall allow users to view and clear diagnostic trouble codes (DTCs). 1.6 The system shall support multi-language options for the user interface.

2. Data Acquisition

2.1 The system shall connect to the vehicle's OBD-II port. 2.2 The system shall retrieve diagnostic trouble codes (DTCs) from the ECU. 2.3 The system shall acquire live sensor data from the vehicle, including:

- 2.3.1 Engine temperature
- 2.3.2 Fuel level
- 2.3.3 Vehicle speed
- 2.3.4 Engine Speed
- 2.3.5 DTC Codes
- 2.3.6 Oil Temperature

3. Data Processing

3.1 The system shall process retrieved data to generate diagnostic reports. 3.2 The system shall identify potential issues based on DTCs. 3.3 The system shall provide recommended actions based on diagnostics.

4. Mobile Connectivity

4.1 The system shall utilize a mobile device as a hotspot for cloud connectivity. 4.2 The system shall upload diagnostic data to the AWS Cloud Server. 4.3 The system shall allow users to manually trigger data uploads to the cloud. 4.4 The system shall support secure data transmission to the cloud.

5. Cloud Integration

5.1 The system shall store historical diagnostic data on the AWS Cloud Server. 5.2 The system shall allow users to retrieve past diagnostics from the cloud. 5.3 The system shall provide users with a summary of past issues and resolutions.

Technical Investigations

1.Raspberry Pi Platform

Purpose: Raspberry Pi will be used as the central processing unit (CPU) for the monitoring screen. It will interface directly with the OBD-II adapter to collect and process the real-time vehicle data.

Justification: Raspberry Pi is ideal for this scenario as it is inexpensive, flexible and has a lot of community support as well as being able to run python and interface with a wide range of hardware components (e.g., Bluetooth, Wi-Fi).

Key Functions: Displaying diagnostics, managing data flow, executing diagnostic logic, and supporting cloud uploads.

2. OBD-II Protocol and Adapter

Purpose: Standard access to vehicle data is made available by the OBD-II protocol. Data will be collected from the vehicle's Electronic Control Unit (ECU) using an OBD-II adaptor that is compatible with Raspberry Pi.

Justification: Access to critical metrics such as speed, RPM, coolant temperature, and diagnostic trouble codes (DTCs) is made possible via the widely used OBD-II protocol for vehicle diagnostics.

Key Functions: enabling data transmission from several sensors to the monitoring system and facilitating the link between the Raspberry Pi and the car's ECU Protocol and Adapter for OBD-II

3. CAN (Controller Area Network) Protocol

Purpose: The vehicle's ECU data will be accessed via the CAN protocol, which is necessary for dependable communication with the OBD-II system.

Justification: OBD-II communication requires CAN, a reliable automotive network protocol. It allows the ECU and Raspberry Pi to exchange real-time diagnostic data.

Key Functions: Enabling low-latency data transfer between the OBD-II adapter and Raspberry Pi, ensuring real-time monitoring.

4. Python Programming Language

Purpose: Python will be used for coding the main application on the Raspberry Pi, handling data collection, processing, display, and cloud integration.

Justification: Python's simplicity, extensive libraries, and community support make it suitable for developing IoT and data monitoring applications.

Key Functions: Handling data acquisition from OBD-II, processing diagnostics, managing UI updates, and supporting cloud communication.

5. AWS Cloud Services

Purpose: AWS will be employed for data storage, enabling historical diagnostics to be accessed, processed, and retrieved.

Justification: AWS provides scalable, secure, and cost-effective cloud storage and data processing capabilities.

Key Functions: Storing and managing historical diagnostic data, providing an interface for past diagnostics retrieval, and allowing secure data synchronization between the vehicle system and the cloud

6. Mobile Connectivity (via Hotspot)

Purpose: A mobile device will provide a hotspot connection to facilitate data uploads from the Raspberry Pi to AWS.

Justification: Many vehicles do not have built-in internet connectivity, so using a mobile hotspot provides an efficient solution for internet access.

Key Functions: Allowing the Raspberry Pi to connect to AWS for data uploads, enabling real-time cloud synchronization, and supporting manual uploads for data backup.

7. Using PyQt/PySide for User Interface (UI) Development

Goal: A graphical user interface that shows real-time vehicle data will be developed using PyQt or PySide, which are Python bindings for the Qt framework.

Justification: With support for widgets like graphs, gauges, and tables, these libraries can be used to create an interactive, customisable interface.

Key Functions: displaying data in an intuitive manner, permitting simple interaction and navigation, and facilitating the visualisation of important variables like temperature, RPM, and speed.

These technologies collectively enable a robust OBD-II data monitoring system that provides real-time vehicle insights, preventative diagnostics, and secure data storage.

Non – Functional Requirements

Performance Requirements

Data Processing Rate: In order to give the user updates almost instantly, the system needs to process sensor data at least 50 times per second. While still feasible with the Raspberry Pi, this will guarantee data accuracy and immediacy.

Data Display lag: There should be no more than 200 milliseconds of lag between data from the car's ECU and the interface. To provide users with instant feedback on vehicle data, this is crucial.

Data Upload Frequency: While the vehicle is operating, diagnostic data uploads to the AWS Cloud Server should happen every five minutes. If necessary, users can choose to manually initiate uploads.

Platform Compatibility

Hardware Platform: To guarantee customer cost-effectiveness and ease of installation, the application needs to work with Raspberry Pi models 3 and 4.

OBD-II Protocol Compatibility: To guarantee compatibility with a variety of automobiles built after 1996, the system must accept the standard OBD-II protocols (SAE J1850, ISO 9141-2, ISO 14230-4, and ISO 15765-4 CAN).

Mobile OS Compatibility: To ensure dependable cloud connectivity across the main mobile platforms, the system needs to connect to mobile devices running iOS 12.0+ and Android 8.0+ for hotspot capability.

Security Requirements

Authentication and Authorisation: To guarantee safe access and stop unwanted data access, all communications with the AWS Cloud Server must be authenticated using OAuth 2.0 and

SSL/TLS encryption.

Data Encryption: To protect user data, diagnostic data saved on AWS must be encrypted in transit over HTTPS and at rest using AES-256.

Access Control: The cloud server's data upload and retrieval features should only be accessible to authorised users who have been verified by the application.

Reliability Requirements

System Uptime: To minimise downtime and guarantee continuous monitoring, the system should operate with an uptime of 99.5%.

Fault Tolerance: By queuing data for upload as soon as connectivity is restored, the system should be built to withstand minor errors (such as a brief loss of internet connection) without crashing.

Usability Requirements

Interface Readability: The user interface should be designed to display clear and readable information, with a minimum font size of 12pt for metrics and a contrast ratio of at least 4.5:1 for visibility during different lighting conditions.

Tutorial Accessibility: The application should include onboarding tutorials and offer video demonstrations on connecting to the OBD-II port and using the interface for novice users.

Scalability Requirements

Data Storage Scalability: AWS Cloud storage must support dynamic scaling to handle an increasing volume of historical data as more diagnostic records are uploaded.

User Scaling: The system should support concurrent use by at least 100 users accessing their data on the cloud without a decrease in performance.

Operational and Environmental Limitations

Operating Temperature: To guarantee functionality in harsh environments, the system—in particular, the Raspberry Pi and OBD-II adapter—must be able to function in a temperature range of -20°C to 70°C.

Power Consumption: Without depleting the battery, the monitoring system ought to run on vehicle power. As soon as the car is stopped moving, it ought to switch to a low-power standby mode.

Compliance Requirements

Industry norms Compliance: To meet industry standards for vehicle diagnostics, the system must adhere to SAE and ISO standards for OBD-II and CAN protocols.

Maintenance Requirements

Software upgrades: To enable smooth upgrades without requiring user involvement, the system should offer over-the-air (OTA) updates for software fixes and enhancements.

Error Logging and Monitoring: The system must log errors locally and have an option to upload error logs to AWS for diagnostic purposes, aiding in remote troubleshooting and maintenance.

Quality Assurance Provisions

Software Test Procedures

The goal is to verify that the system is error-free, all components work as intended, and the program performs as intended.

Unit Testing

Purpose: To test individual components of the system, such as data acquisition, data processing, and user interface modules.

Tools:

Pytest: For testing Python functions and modules on the Raspberry Pi and verifying expected outputs for sensor readings, connectivity, and data transformations.

Using testing rigs from college to simulate OBD-II Data

Procedure:

Test each function, such as acquiring sensor data (RPM, coolant temperature) or retrieving DTCs, with both expected and boundary inputs.

Mock AWS and mobile hotspot connections to test data uploads.

Integration Testing

Purpose: To verify that individual components (e.g., Raspberry Pi, OBD-II adapter, AWS server) work together as expected.

Tools:

Docker: For setting up test environments that replicate the AWS backend.

Continuous Integration (CI) tools (e.g., GitHub Actions): Automates integration tests with each code update.

Procedure:

Test the data flow from the vehicle's sensors to the Raspberry Pi and onwards to AWS, including the interaction between Raspberry Pi software, the OBD-II adapter, and cloud storage.

Perform tests in different vehicle conditions (e.g., idle, moving, high RPM) to ensure consistent data transmission and reliability.

System Testing

Purpose: To test the complete system, verifying end-to-end functionality from data collection through display to cloud uploads.

Tools:

Selenium: For automated UI testing, verifying that real-time data is displayed correctly on the interface and that interactions, such as clearing DTCs, function as intended.

JMeter: For load testing, simulating multiple users accessing AWS data simultaneously to ensure stable performance.

Procedure:

Perform tests with the entire system in a simulated vehicle environment, validating that each feature (e.g., live data display, manual data upload) works as expected.

Stress test AWS data retrieval under high load to confirm cloud service resilience.

Performance Testing

Purpose: To confirm that the system meets performance standards, such as data processing rate, display latency, and cloud upload intervals.

Tools:

Apache JMeter: For load testing data upload and retrieval from AWS, simulating multiple users.

Pi-stress: A Raspberry Pi utility for CPU and memory load testing to ensure the system can handle peak data processing without latency.

Procedure:

Measure the system's data processing rate and confirm it meets the minimum requirement of 50 times per second.

Test data upload intervals and system responsiveness under both low and high vehicle activity.

Security Testing

Purpose: To ensure the system's security standards are met, including secure data transmission and proper authentication.

Tools: OWASP ZAP: For testing vulnerabilities in web applications, particularly around the AWS cloud integration.

SSL Labs: For checking the security level of SSL/TLS implementations.

Procedure: Test HTTPS connections between the Raspberry Pi and AWS to confirm data is encrypted in transit.

Conduct vulnerability scans to detect potential security risks, such as insecure communication or weak authentication methods.

Software Validation Procedures

Objective: To validate that the software meets user needs and performs reliably in realistic conditions. This stage will involve user feedback, real-world testing, and continuous evaluation.

User Acceptance Testing (UAT)

Purpose: To validate that the system aligns with user expectations, confirming that the interface, data display, and functionality are intuitive and meet user needs.

Procedure:

Conduct user testing with potential end-users, such as vehicle owners and enthusiasts.

Collect feedback on ease of use, clarity of real-time data displays, and responsiveness of controls (e.g., clearing DTCs).

Make iterative adjustments based on feedback to enhance usability.

Field Testing (Real Vehicle Testing)

Purpose: To validate the system under actual driving conditions and confirm that it functions as expected in various scenarios.

Procedure:

Install the system in several vehicles and test in various conditions, such as city driving, highway driving, and idle states.

Monitor the system's ability to accurately and consistently display real-time data, detect DTCs, and upload data to AWS during actual use.

Evaluate reliability over extended operation times to identify any power or overheating issues.

Regression Testing

Purpose: To ensure that new updates or patches do not negatively impact the existing functionality of the software.

Tools:

Automated Test Suite with Pytest and Selenium: To rerun tests on core functionalities after each software update.

Procedure:

Perform automated regression tests after each software update to ensure that real-time monitoring, data upload, and interface functions continue to work as expected.

Address any issues identified during regression to prevent disruptions in functionality.

Beta Testing

Purpose: To test the system with a wider user base in a "real-world" setting and gather feedback on system performance, usability, and reliability.

Procedure:

Distribute the system to a small group of beta users who drive different vehicle types and have varying technical proficiency.

Collect feedback on overall satisfaction, usability issues, and any bugs encountered.

Use the results to make final adjustments to ensure the system meets user expectations before full release.

These quality assurance provisions establish a comprehensive approach to testing and validating the OBD-II data monitoring system, ensuring that it performs as specified, meets user needs, and functions reliably in diverse conditions.

Methodology Overview: Agile Development

Overview: Agile development prioritises iterative work cycles, or "sprints," with each sprint concentrating on a particular set of enhancements or features. At the conclusion of every sprint, feedback is obtained, allowing for ongoing improvement and modification to satisfy user requirements and quickly resolve issues. This iterative method works well for evaluating device integrations and adjusting in response to user feedback.

Agile Stages and Activities

1. Planning & Requirements Gathering

- Activities:
    - Define Project Scope: Outline the high-level goals of the OBD-II monitoring system, focusing on real-time data display, diagnostics, and cloud storage.
    - Backlog Development: Create a backlog of features, prioritized based on user value and development dependencies (e.g., core features like data acquisition before advanced features like historical data retrieval).
    - Sprint Planning: Define the scope for the initial sprints, focusing on setting up essential hardware interfaces and basic UI functionalities.

2. Design Phase (Sprint 1)

- Activities:
    - System Architecture Design: Map out the architecture, including the interaction between the OBD-II adapter, Raspberry Pi, mobile device, and AWS.
    - Interface Mock-ups: Create initial UI designs and wireframes for the display, including basic layout for live data monitoring, error code display, and cloud synchronization options.
    - Database and Data Flow Diagrams: Define the data structures, such as how data will be organized on AWS, and detail data flow from the vehicle's sensors to the Raspberry Pi and cloud storage.

3. Development & Iterative Sprints

- Activities for Each Sprint:
    - Sprint 2: Hardware Integration
        - Set up and test the connection between the Raspberry Pi and the OBD-II adapter via Bluetooth or USB.
        - Retrieve basic sensor data (e.g., RPM, coolant temperature) from the vehicle's ECU to ensure accurate data collection.
        - Test data accuracy and resolve any connectivity or compatibility issues with the OBD-II adapter.
    - Sprint 3: Basic UI and Real-Time Data Display

        - Develop the user interface to display real-time data (e.g., RPM, coolant temperature).
        - Test the update rate to meet real-time display requirements.
        - Integrate basic navigation for users to switch between views, such as a summary dashboard and individual metrics.
    - Sprint 4: Cloud Integration and Data Upload
        - Implement data upload to AWS through a mobile device acting as a hotspot.
        - Ensure secure data transmission and test for consistent data synchronization.
        - Create a basic view for historical data retrieval and display past diagnostics.
    - Sprint 5: Error Detection and Notifications
        - Implement functionality to detect and display Diagnostic Trouble Codes (DTCs) based on sensor data.
        - Enable options for users to clear error codes from the interface.
        - Test user notification features (e.g., alerts for critical engine temperature).
    - Sprint 6: Advanced Features and User Interface Enhancements
        - Add graphical elements, such as gauges and graphs, for more intuitive real-time monitoring.
        - Implement language support for multi-language interfaces as outlined in requirements.
        - Include final enhancements, such as system preferences and user customization options.

4. Testing Phase

- Activities:
    - Unit Testing: Conduct unit tests for each function, such as reading RPM data, connecting to AWS, or displaying error codes.
    - Integration Testing: Validate that components (Raspberry Pi, OBD-II adapter, cloud services) work together seamlessly by testing data flow end-to-end.
    - System Testing: Perform system-wide tests, ensuring that all features, including data display, cloud upload, and user interface, work as expected.
    - User Acceptance Testing (UAT): Gather feedback from a small group of beta users (e.g., car enthusiasts or mechanics) to identify usability issues and areas for improvement.

5. Deployment Phase

- Activities:
    - Final Debugging and Optimization: Address any performance bottlenecks, such as data refresh rate or power consumption, to ensure a smooth user experience.
    - Documentation: Create user manuals, installation guides, and troubleshooting instructions for users.
    - Initial Release: Deploy the system for use, making it available to the intended user group.
    - Feedback Collection: Gather user feedback post-deployment to identify areas for further refinement.

6. Maintenance & Iteration

- Activities:
    - Continuous Monitoring: Monitor user feedback and system performance, particularly with cloud data synchronization and real-time display.
    - Ongoing Bug Fixes and Updates: Address bugs, update software as needed, and add new features or improvements based on user feedback.
    - Periodic Feature Enhancements: Release updates, such as adding more sensor integrations or enhancing UI customization options, based on user needs and feedback.

Benefits of Agile for This Project

- Flexibility: Agile allows rapid adjustments based on testing feedback, especially critical for hardware-software integration.
- User-Centric Development: Agile prioritizes delivering valuable features iteratively, ensuring that the system meets user needs with each release.
- Risk Reduction: The iterative approach helps to identify and mitigate risks early, such as hardware compatibility issues or data synchronization challenges.

Risks and Contingencies

Developing a real-time OBD-II monitoring system poses risks including as hardware compatibility and user adoption concerns, which could hinder project success. Here's an overview of important dangers and their associated contingency plans:

1. Hardware Compatibility Issues

Risk: The OBD-II adapter may not be compatible with all vehicles or may fail to retrieve data from certain vehicle models, especially older or specialized models.

Contingency Plan:

Compatibility Testing: Conduct thorough testing on a diverse set of vehicles early in development to identify compatibility issues.

Adapter Alternatives: Prepare a list of alternative OBD-II adapters compatible with a broader range of vehicles. If the primary adapter fails, these alternatives can be used.

User Documentation: Provide a list of compatible vehicle models and adapters in the user documentation to manage user expectations.

2. Real-Time Data Processing Delays

Risk: The Raspberry Pi may struggle to process data quickly enough for real-time display, particularly if sensor data frequency is high or additional processing is required.

Contingency Plan:

Optimize Code: Focus on optimizing the code to improve data processing efficiency, using streamlined algorithms and reducing unnecessary data processing.

Hardware Upgrade: Plan for the potential to upgrade to a more powerful Raspberry Pi model or another single-board computer if needed to handle processing requirements.

Display Rate Adjustment: Allow users to adjust the data refresh rate on the display, trading off update speed for smoother performance on lower-powered devices.

3. Data Security and Privacy Concerns

Risk: Sending vehicle data to the cloud may expose critical information to security dangers, such as data interception.

Contingency Plans:

Encryption: Use end-to-end encryption for data transfers between the Raspberry Pi and AWS to secure data in transit.

Access Control: Prevent unauthorised access to cloud data by implementing strong access controls and authentication requirements.

User Awareness: Inform users of data privacy measures and allow them to opt-out of cloud storage if they prefer local-only storage.

4. Connectivity Issues for Cloud Integration

Risk: Data uploads may fail or be delayed due to reliance on a mobile hotspot connection, especially in areas with poor cellular service.

Contingency Plan:

Offline option: Enable the offline option, which stores data locally on the Raspberry Pi and uploads it to the cloud when a connection is available.

Error Handling: Implement error-handling protocols to automatically retry uploads whenever connectivity is restored.

Data Compression: Reduce the quantity of data transferred by compressing it before uploading, which reduces upload times and the need for high-speed connections.

5. Power Supply Constraints

Risk: The Raspberry Pi may drain the vehicle’s battery if it remains active for extended periods without the vehicle engine running.

Contingency Plan:

Automated Shut-off: Create an automated shut-off function that turns off the Raspberry Pi after a certain time of inactivity or when the vehicle's engine is turned off.

External Power Source Option: Allow users to connect an external power bank if needed for extended use while the engine is off.

Low-Power Mode: Implement a low-power mode that reduces processing and display frequency when the vehicle is idle.

6. Unexpected Sensor Data Variability

- Risk: Sensor readings may vary unexpectedly across vehicle models or under certain conditions, affecting data accuracy.
- Contingency Plan:
    - Calibration Feature: Include a calibration feature to allow users to adjust readings based on their specific vehicle and environment.
    - Data Smoothing Algorithms: Use data smoothing or filtering algorithms to reduce erratic or fluctuating sensor values, ensuring a more consistent user experience.
    - Data Validation Checks: Implement checks to flag and manage suspicious or extreme values that fall outside expected ranges.

7. Software Bugs and System Failures

Risk: Software bugs could lead to crashes, data loss, or incorrect diagnostics, impacting the system's reliability.

Contingency Plan:

Automated Testing: Use automated tests to detect bugs early in development and after each update to limit the chance of introducing new problems.

To reduce data loss, often backup user data to local storage before transmitting it to the cloud.

Fail-Safe Mechanism: Create a fail-safe mechanism that reboots the system if a critical fault occurs, reducing downtime while maintaining functioning.

8. High User Interface Complexity

Risk: The system's user interface could be challenging for users with limited technical expertise, reducing user adoption.

Contingency Plan:

Usability Testing: Conduct frequent usability testing with a diverse range of users to discover and resolve areas of misunderstanding or difficulty.

Simplified Mode Option: Create a simplified mode that includes only the most important elements, allowing users to access basic functionalities without having to navigate complex settings.

Video Tutorials and Documentation: Provide simple video tutorials and written documentation to help users with setup, troubleshooting, and interface navigation.

9. Unclear Diagnostic Information for Users

Risk: The OBD-II system may display codes or diagnostic information that users find difficult to interpret.

Contingency Plan:

In-App Explanations: Integrate a help section that explains common Diagnostic Trouble Codes (DTCs) and their meanings, assisting users in understanding issues without needing external resources.