FixIT

Design Document

sdmay25-44
Berk Gulmezoglu
Benjamin Muslic: Embedded Engineer
Mohamed Elaagip: Front-End Engineer and UI Designer
Jonathan Duron: Front-End Engineer
William Griner: Backend and Cloud Engineer
Thecarcloudproject@gmail.com

https://sdmay25-44.sd.ece.iastate.edu

Executive Summary

Modern vehicles generate Diagnostic Trouble Codes (DTCs) when issues arise, but interpreting these codes remains a significant challenge for most drivers. This knowledge gap often leads to unnecessary expenses and uncertainty about vehicle repairs. FixIt addresses this problem by developing an intelligent mobile application that translates complex DTCs into easily understandable information.

Key design requirements include:

- OBD-ll reader compatibility with all compliant vehicles
- ESP32 communication via Bluetooth for sending data
- AI-based code interpretation from community forums
- Real-time diagnostics
- User friendly interface
- Cost-effective solution

This design uses multiple technologies and approaches:

- ESP32 microcontroller for OBD-ll communication
- React Native for cross platform mobile development
- Bluetooth Low Energy (BLE) for communicating to devices
- Cloud services for data processing

Our team's current progress:

- Successfully established ESP32 communication with OBD-ll scanner
- Developed and tested prototype mobile application
- Implemented Bluetooth communication between ESP32 and mobile app
- Created user interface for DTC's display and interpretation

Learning Summary

Development Standards & Practices Used

Hardware Standards:

- OBD-II Protocol Standards (SAE J1979)
- ISO 15765-4 (CAN) communication protocol
- ISO 9141-2 protocol support
- Bluetooth Low Energy (BLE) specifications

Software Development Standards:

- React Native development guidelines
- TypeScript coding standards
- Git version control practices
- ESP32 programming standards
- Mobile app UI/UX design principles

Testing Equipment:

- ECUsim 2000 emulator for OBD-II simulation
- ISO protocols (9141-2, 14230-4, 15765-4)
- Fault simulation capabilities (SF)

Engineering Standards:

- CAN bus communication standards
- ESP32 hardware specifications
- Bluetooth 5.0 specifications
- OBD-II pin configuration standards
- USB communication protocols

Summary of Requirements

Functional Requirements:

- OBD-II reader compatibility with all compliant vehicles
- ESP32 communication via Bluetooth for data transmission
- Real-time diagnostics and monitoring
- Vehicle health monitoring capabilities

Physical Requirements:

- Hardware portability (OBD-II reader and ESP32)
- Wi-Fi and cloud integration

- ECUsim2000 for development and testing

Technical Requirements:

- OBD-II protocol support

User Interface Requirements:

- Clean, modern, and intuitive layout
- Easy-to-understand icons
- Clear navigation structure
- Real-time data display capabilities

User Experience Requirements:

- Easy device setup and pairing
- Fast data processing and display
- Seamless cloud integration
- Secure data transmission
- Clear error code interpretation
  Economic Requirements:
- Total hardware cost under $100
- Cost-effective solution for DIY users

Applicable Courses from Iowa State University Curriculum

COM S 319: Software Construction and User Interfaces

- React Native mobile application development
- User interface design for diagnostic display
- Component-based architecture implementation

COM S 309: Software Development Practices

- Version control and collaborative development
- API integration for diagnostic data
- Software testing methodologies Hardware and Protocols

CPR E 288: Embedded Systems

- ESP32 microcontroller programming
- Protocol implementation (ISO 15765-4, ISO 9141-2)
- Hardware-Software integration

CPR E 489: Computer Networking and Data Communications

- Bluetooth communication protocols
- Data transmission and error handling
- Network security implementation

CPR E 490: Senior Design

- Project management and documentation
- System integration
- Requirements analysis and validation
- Testing and verification procedures

New Skills/Knowledge acquired that was not taught in courses

Development Technologies

- React Native mobile development
- Bluetooth Low Energy (BLE) implementation
- Cross-platform mobile application development

Automotive Protocols

- OBD-II diagnostic protocols
- ISO 15765-4 (CAN) communication
- Multiple protocol handling (ISO 9141-2, ISO 14230-4)

Testing Tools

- ECUsim 2000 emulator usage
- Protocol verification techniques
- Fault simulation procedures

Hardware Integration

- ESP32 Bluetooth configuration
- OBD-II interface programming
- Multi-protocol hardware communication

Development Practices

- Mobile app state management
- Real-time data handling
- Protocol-specific error handling
- Cross-platform deployment strategies

Table of Contents

1. Introduction

1.1. Problem Statement

Modern vehicles are equipped with onboard diagnostic systems that generate codes when something goes wrong, but for most drivers, understanding these Diagnostic Trouble Codes (DTCs) is a frustrating challenge. These codes are often cryptic, lacking sufficient explanation or context. Drivers are left scouring the internet for answers or relying on mechanics, which can result in costly repairs or potential overcharging due to a lack of understanding.

In today's fast-paced world, people don't have time to research every potential issue with their car, and many are unsure if they're receiving an honest assessment from their mechanic. This knowledge gap affects both seasoned DIY enthusiasts who want to repair their own vehicles and everyday drivers who just want to avoid unnecessary expenses.

FixIt is here to solve this problem. Our AI-driven app translates those confusing DTC codes into easy-to-understand insights by gathering information from a wide range of trusted online sources and communities. It gives users the full story of what's wrong with their vehicle, right at their fingertips, empowering them to make informed decisions, whether they're performing the repair themselves or taking the car to a professional. In doing so, FixIt not only saves users time and money but also provides peace of mind, protecting them from inflated repair costs.

1.2. Intended Users

DIY Car Enthusiasts:

These are car owners who enjoy working on their vehicles, often performing maintenance and repairs on their own. They are typically knowledgeable about car systems but may not always understand the complexity behind modern diagnostic codes. They value tools that enhance their ability to maintain their cars independently.

- Needs: DIY car enthusiasts need a tool that translates complex DTC codes into actionable insights, allowing them to quickly diagnose issues without sifting through multiple online forums or manuals. They want to save money and avoid relying on mechanics for every repair.
- Benefits: FixIt empowers DIYers by providing clear, accurate explanations of DTC codes, drawn from AI-sourced community insights. By simplifying this process, FixIt saves them time and ensures they have all the information they need to perform repairs efficiently. This connects to the broader problem of complex diagnostic codes and eliminates the hassle of researching each issue.

Everyday Car Owners

This group includes people who rely on their vehicles for daily transportation but have little technical knowledge about cars. They typically rely on mechanics for maintenance and repairs and may be susceptible to overcharging or unnecessary services due to a lack of understanding.

- Needs: Everyday car owners need a tool that demystifies car diagnostics, enabling them to understand what's going wrong with their vehicle without needing to be mechanically inclined. They want peace of mind and confidence when discussing repairs with mechanics, knowing they're not being taken advantage of.
- Benefits: FixIt ensures these users get straightforward, reliable information about their car's issues. By doing so, the app reduces the risk of being overcharged by unscrupulous mechanics and enables them to make informed decisions about repairs. This value aligns with the problem of mechanic scams and helps to protect users from inflated repair costs, giving them more control over their car maintenance.

Car Sellers

Car sellers, whether individuals or small dealerships, need to keep their vehicles in good working order to sell them for the best price. They may want to diagnose and fix issues themselves or at least understand the vehicle's condition to communicate effectively with potential buyers.

- Needs: Sellers need a reliable and quick way to assess the health of a vehicle before putting it up for sale. They want to identify and fix any issues beforehand to maximize the sale price or provide potential buyers with accurate diagnostic information to build trust.
- Benefits: FixIt helps car sellers by quickly identifying problems and offering solutions, ensuring their vehicle is in top condition for sale. It also helps them provide detailed diagnostics to buyers, increasing the likelihood of a successful sale by offering transparency. This addresses the broader issue of ambiguous diagnostic codes and builds trust between seller and buyer, fostering smoother transactions.

Connection to Problem Statement:

All these user groups—DIYers, everyday car owners and car sellers —benefit from FixIt because it tackles the core problem of DTC code complexity. By transforming confusing diagnostic information into clear, actionable insights, FixIt saves time, money, and effort for all users while ensuring transparency and preventing mechanic fixes quote scams. The app bridges the gap between professional-level diagnostics and everyday car maintenance, ensuring that users are equipped with the knowledge they need to make informed decisions about their vehicles.

2. Requirements, Constraints, And Standards

2.1. Requirements & Constraints

Functional Requirements:

OBD-II Reader Compatibility:

- The system must be able to read DTC codes from all OBD-II compliant vehicles.
- Constraint: OBD-II compatibility required for vehicles manufactured after 1996).

ESP32 Communication:

- The ESP32 must establish a stable connection with the ELM327 via Wi-FI and the user's phone via Bluetooth

AI-Based Code Interpretation:

- The system must interpret DTC codes using AI to gather insights from community forums, databases, and other online resources.

Real-Time Diagnostics:

- The app must provide real-time analysis of car issues, displaying results as soon as codes are read from the vehicle.

Vehicle Health Monitoring:

- The system should live data such as temperature

Physical & Resource Requirements:

Hardware Portability:

- The OBD-II reader must be small and portable

ESP32 Power Supply:

- The ESP32 must be powered by the car's OBD-II port or an internal battery, eliminating the need for external power sources.

Aesthetic Requirements:

User-Friendly App Design:

- The app's interface must be aesthetically pleasing with a clean, modern, and intuitive layout, incorporating easily recognizable icons and color schemes for different car statuses (e.g., green for good, red for urgent).

Intuitive App Design:

- The app's interface must be easy for the user to understand and navigate.

User Experiential Requirements:

Easy Setup:

- The OBD-II dongle must be easy to setup.

Fast Data Processing:

- The system must send DTC data to the cloud and return results within a few moments

Seamless User Experience:

- The user experience should be seamless, with the app automatically receiving results from the cloud once the ESP32 has sent diagnostic data.

Economic Requirements:

Cost-Effective Solution:

- The Wi-Fi OBD dongle and ESP32 system must be affordable for DIY users and casual car owners.
- Constraint: Total hardware cost must not exceed $100

2.2. Engineering Standards

Software life cycle processes - Maintenance

IEEE/ISO/IEC 14764-2021
ISO/IEC/IEEE 14764 establishes a comprehensive framework for software maintenance throughout a system's lifecycle, recognizing maintenance as a vital component that consumes substantial resources. The standard covers both pre-delivery activities (like planning and logistics) and post-delivery activities (such as help desk support and upgrades). The standard defines four main types of maintenance:

- Corrective: Fixing identified problems
- Adaptive: Maintaining software usability as environments change
- Preventive: Addressing potential issues proactively
- Perfective: Enhancing performance and maintainability

The process framework encompasses strategy development, planning, problem analysis, modification implementation, and quality assurance. It provides guidance for both internal teams and external providers, addressing everything from minor updates to major modifications. The standard emphasizes careful planning and execution, particularly for systems that must maintain continuous operation. Key aspects include comprehensive quality assurance measures, thorough testing requirements, and robust configuration management practices. The standard mandates proper documentation of all maintenance activities, including problem reports, modification requests, and implementation details. It establishes clear procedures for change control to maintain system stability and reliability throughout the maintenance process.

Software testing -- Part 2: Test processes

IEEE/ISO/IEC 29119-2-2021
This international standard (ISO/IEC/IEEE 29119-2) establishes a comprehensive framework for software testing processes that can be applied across any organization or software development project. It provides a structured approach to testing through three main process layers: organizational test processes, test management processes, and dynamic test processes. The standard aims to help organizations standardize and improve their testing practices by defining clear processes, roles, and deliverables.

The standard is designed to be flexible and adaptable, allowing organizations to either fully implement all processes or tailor them to their specific needs. It emphasizes risk-based testing approaches, helping organizations prioritize testing efforts based on identified risks and potential impacts. The framework supports both traditional and agile development methodologies and can accommodate various types of testing, including manual, automated, functional, and nonfunctional testing.

A key goal of the standard is to provide those responsible for software testing with the necessary information and structure to effectively manage and perform testing activities across their organization. It accomplishes this by defining specific process requirements, outcomes, and activities while allowing for different levels of conformance. The standard also aligns with other important software development and quality standards, making it easier for organizations to integrate it into their existing processes.

The standard's practical value lies in its ability to help organizations establish consistent testing practices, improve test planning and execution, and better manage testing resources and activities. By following these standardized processes, organizations can potentially reduce testing risks, improve software quality, and create more predictable testing outcomes.

Systems and software assurance -- Part 4: Assurance in the life cycle

IEEE/ISO/IEC 15026-4-2021

This standard (ISO/IEC/IEEE 15026-4:2021) provides guidance on how to ensure and demonstrate that critical system and software properties meet their required levels of assurance throughout their lifecycle. It focuses on two main aspects: achieving specific assurance claims about a system or software, and proving that these claims have been met through proper documentation and evidence. The standard introduces two key process views - one for systems and one for software - that work alongside existing lifecycle process standards (ISO/IEC/IEEE 15288 and 12207). These views help organizations integrate assurance activities into their regular development and maintenance processes. The standard explains how to identify assurance claims (critical properties

that need to be guaranteed), gather evidence to support these claims, and construct valid arguments demonstrating that the claims have been achieved.

The document is designed to be used in various contexts: for agreements between suppliers and acquirers, for regulatory compliance, or for internal development process improvement. It provides specific guidance on how to manage assurance-related activities throughout the project lifecycle, including planning, risk management, configuration control, and quality assurance.

What makes this standard particularly valuable is its focus on maintaining assurance throughout the entire system or software lifecycle - not just during initial development. It emphasizes the importance of documenting and maintaining assurance information, managing changes that might affect assurance claims, and ensuring that critical properties continue to be met even as systems evolve over time.

The standard serves as a practical guide for organizations needing to demonstrate that their systems or software meet specific critical requirements, particularly in areas where failure could have serious consequences, such as safety, security, or reliability.

3 Project Plan

3.1 Project Management/Tracking Procedures

Project Management Style:

We adopted a hybrid **Waterfall + Agile** approach. The Waterfall model helped us outline the full scope, requirements, and design documentation early on, which aligned with university deadlines and deliverables. Agile allowed us to iterate on implementation, especially on the mobile app and backend features, through short development sprints.

Tools Used:

- **GitHub**: Version control, pull requests, and task branches
- **AWS**: Used to have a cloud environment where we can store and process data
- **Discord**: Daily communication and quick updates
- **Weekly meetings**: In-person/virtual syncs for demoing progress, planning sprints, and assigning tasks

3.2 Task Decomposition

To effectively manage the development of FixIT, we decomposed the project into major technical areas, each consisting of multiple interdependent subtasks. This structure allowed us to allocate

responsibilities, estimate effort, and track progress through Agile sprints while maintaining high-level project deliverables.

1. Embedded Hardware Integration (ESP32 + OBD-II)

Lead: Benjamin Muslic

- Develop and flash ESP32 firmware for OBD-II communication
- Implement support for ISO 9141-2, ISO 14230-4, and ISO 15765-4 (CAN) protocols
- Write code to parse DTCs and sensor data from the ECU
- Configure BLE advertisement and connection logic
- Transmit structured data packets via BLE to mobile devices
- Validate functionality using the ECUsim 2000 OBD-II simulator

2. Bluetooth Communication Layer

Collaborative: Ben, Mohamed, Jonathan

- Implement BLE pairing and connection on ESP32 and mobile app
- Design and test custom BLE data packet schema
- Handle BLE reconnection, disconnection, and OS-specific behavior
- Ensure reliable cross-platform BLE communication on Android and iOS
- Ensured only relevant BLE advertisement displayed

3. Mobile Application Development (React Native)

Leads: Mohamed Elaagip, Jonathan Duron

- Design and implement intuitive UI/UX for diagnostics and vehicle health display
- Integrate BLE client to receive and parse ESP32 data
- Handle navigation, app state, and maintenance tracking
- Display raw and interpreted DTCs with urgency and guidance
- Develop premium features like multiple vehicle tracking and report generation
- Developed a chatbot where users can ask questions about their vehicles and make recommendations to the user based off the information the user inputs
- Integrated options for users to input more than one vehicle they own

4. Backend Infrastructure & Cloud Services

Lead: William Griner

- Build HTTP API endpoints that integrate with AWS Lambda (Python)
- Store and retrieve user information using a PostgreSQL database
- Utilize Infrastructure-as-Code (IaC) to enable version control and transparency in our cloud infrastructure
- Use S3 and DynamoDB for infrastructure state locking, in order to prevent multiple engineers from modifying our cloud infrastructure in contradicting ways at the same time.

- Optimize infrastructure for fault tolerance by using multiple availability zones.

5. AI-Based DTC Code Interpretation

Collaborative: All

- Format prompts for sending DTCs to AI API
- Parse and display returned explanations, causes, and urgency levels
- Handle null, ambiguous, or delayed responses gracefully
- Refine prompt structure to improve clarity and accuracy of interpretations

6. Testing & Validation

Collaborative: All

- Unit test ESP32 firmware with simulated ECU responses
- Test BLE data flow under multiple phone OS environments
- Conduct integration testing between all subsystems
- Perform usability tests with non-technical users
- Run system-level tests using ECUsim and test vehicles
- Track and resolve bugs, regression-test after each update
- Tested navigation across all screens

7. Documentation & Reporting

Lead: Mohamed Elaagip

- Write and maintain the design document, final report, and operation manual
- Prepare testing documentation and configuration notes
- Create system architecture diagrams and setup instructions
- Document key design decisions, trade-offs, and user feedback
- Contribute to final presentation and poster for senior design showcase

3.3 Project Proposed Milestones, Metrics, and Evaluation Criteria

1. Hardware Communication

- Milestone: Achieve reliable OBD-II data reading and transmission
- Metric: 80% success rate in OBD-II data reading within the first 3 months
- Evaluation Criteria:
- Test with at least 1 vehicle or simulator (e.g., ECUsim 2000)
- Conduct 10 consecutive read attempts
- Measure and record success rate of data transmission to ESP32

2. Cloud Infrastructure

A) Networking Setup

- Milestone: Implement a virtual private cloud to construct cloud resources in.
- Metric: Be able to create all of the cloud resources we want to create (Lambda, RDS, Internet gateway) in our VPC. The VPC and all of the cloud resources should be created with infrastructure as code. This is for consistency in tracking infrastructure changes and better organization (compared to creating resources through the AWS console).
- Evaluation Criteria: Create our VPC, and all of the cloud resources we want to create inside our VPC, using infrastructure as code.

b) Cloud security

- Milestone: Implement Security Groups to establish least-privilege access principles across our cloud resources, improving cloud security.
- Metric: Create relevant security groups that allow cloud resources to interact with each other as necessary.
- Evaluation Criteria:
    - Resources that are meant to interact with each other can interact with each other via proper security group configuration (i.e. our lambda being able to interact with our database). Resources that should not be able to interact with each other, cannot.

c) Implementation

- Milestone: Integrate the cloud resources we've created with our mobile application.
- Metric: Our mobile application can send data to and retrieve data from our cloud infrastructure.
- Evaluation Criteria: Data is seamlessly passed to and from our cloud infrastructure via our mobile application.

3. Frontend App Development

- Evaluation Criteria: well-structured code, fast page load times, responsive design
- Measure task completion rate (connect device, view DTC, see result)
- Milestone: Complete MVP mobile app with core features
- Metric: 60% user satisfaction in usability testing
- Conduct usability tests with at least 5 users
- Collect feedback through surveys

4. AI Diagnostic Feature

- Milestone: Implement basic AI-powered DTC interpretation
- Metric: Achieve 70% accuracy on common DTCs
- Evaluation Criteria:
- Test with at least 50 known DTC codes
- Compare AI responses to verified explanations
- Measure number of correct, helpful responses

3.4 Project Timeline/Schedule

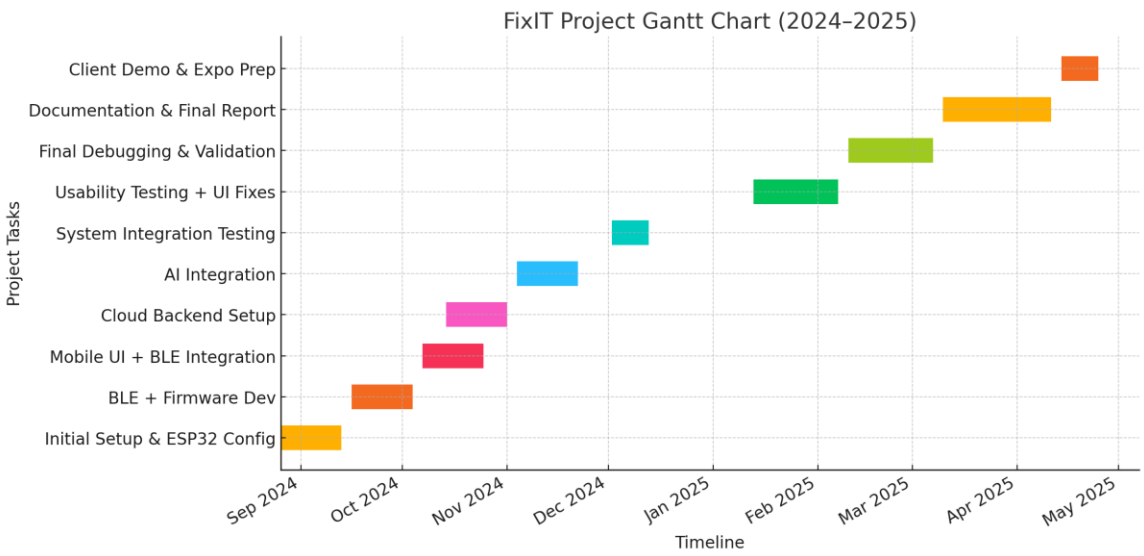


Figure 1.0

The Gantt chart above outlines the planned and actual development timeline for the FixIT project across the Fall 2024 and Spring 2025 semesters. The project was divided into ten major phases, each representing a core component of the system, such as embedded hardware setup, Bluetooth communication, mobile UI development, backend infrastructure, AI integration, and final deployment preparation.

Tasks were scheduled sequentially but with intentional overlaps to allow parallel development where feasible. For example, **Cloud Backend Setup** overlapped with **Mobile UI Integration** to support early testing of API calls, and **Usability Testing** began while **AI Integration** was being finalized to accelerate feedback incorporation.

3.5 Risks and Risk Management/Mitigation

Throughout the development of FixIT, our team identified several potential risks that could impact functionality, performance, or timeline. Each risk was assessed for likelihood and severity, and mitigation strategies were prepared accordingly. We also tracked which risks actually occurred and how they were addressed.

1. BLE Instability Across Devices

Risk: BLE communication behaves inconsistently on different Android and iOS devices

Probability: Medium (0.5)

Mitigation: Test across multiple phone models and OS versions, implement reconnection and fallback logic

Outcome: This risk occurred. We experienced disconnections and delays, especially in background mode. We implemented retry logic and optimized the BLE data flow for stability.

2. AI Interpretation Quality

Risk: AI-generated DTC interpretations may be inaccurate, vague, or slow to return

Probability: High (0.7)

Mitigation: Add timeout handling and fallback responses, refine prompts for clarity

Outcome: This risk occurred. Some AI outputs were unclear or incomplete. We adjusted prompts and added user-friendly fallback text when responses failed.

3. Cloud Function Limitations

Risk: AWS Lambda timeouts or permission issues with database access

Probability: Medium (0.5)

Mitigation: Monitor usage metrics, write efficient API logic, use logging to assist in debugging failures.

Outcome: This risk occurred. We had trouble connecting Lambda to DynamoDB via Terraform. As a temporary fix, we manually configured permissions via the AWS Console. We also did have some errors with Lambda timeouts, but we easily fixed those by increasing the lambda timeout limit.

4. Hardware Parsing Errors

Risk: ESP32 may fail to interpret certain OBD-II responses or protocols

Probability: Medium (0.4)

Mitigation: Support multiple protocols (ISO 9141, CAN, KWP2000), test with ECUsim 2000

Outcome: Minor issues were encountered and resolved through debugging and protocol adjustment.

5. Scheduling Conflicts or Burnout

Risk: Team availability may fluctuate due to academic workload

Probability: Medium (0.6)

Mitigation: Hold weekly meetings, maintain GitHub issues/tasks, redistribute work if needed

Outcome: This risk partially occurred. Team members had schedule conflicts during midterms, so tasks were adjusted to balance the workload and maintain momentum.

6. Usability Challenges for Non-Technical Users

Risk: Users may struggle to pair the device or understand diagnostic results

Probability: Medium (0.5)

Mitigation: Conduct usability testing and iterate on UI/UX design

Outcome: This risk occurred. User testing revealed some confusing UI elements, which we improved based on feedback.

7. Terraform State Conflicts

Risk: Multiple team members running Terraform could corrupt infrastructure state

Probability: Medium (0.4)

Mitigation: Implement state locking using AWS S3 and DynamoDB

Outcome: This risk was successfully mitigated with our locking setup. No major state conflicts occurred.

3.6 Personnel Effort Requirements

The table below outlines the estimated effort required by each team member across both semesters of the project. Responsibilities are distributed based on individual strengths and roles. Tasks were divided into front-end development, backend and infrastructure, hardware integration, AI, testing, and documentation. The effort was planned to ensure consistent contribution from all team members and alignment with project milestones.

First Semester Tasks

Task	Assigned Team Member(s)	Est. Hours
Research which cloud resources to use in areas where we have multiple options (compute, database, API type)	Will	20
Setup of cloud development infrastructure (creation of a VPC, creation of cloud resources, proper security groups)	Will	30
Create database schemas and queries for storing diagnostic insights	Will	6

Setting up bare react native app	Jonathan	10
Allowing BLE connectivity and receiving DTC's	Jonathan	13
Setting up and learning ECU simulator	Jonathan	5
UI/UX wireframes and visual prototyping	Mohamed	12
Hardware setup and ESP32 configuration	Ben	22
ESP32 Programming for ECU data retrieval	Ben	40+
Initial app development and service integration	Mohamed and Jonathan	40
Real-time notification and alert system	Jonathan	10
ESP32 debugging and feature addition	Ben	40+
Hardware-in-vehicle testing for live validation	Ben	15
Debugging frontend/backend as new features roll out	Mohamed and Jonathan	20
Initial integration validation	Ben and Jonathan	10
Technical documentation and sprint reporting	Jonathan	15

Second Semester Tasks

Task	**Assigned Team Member(s)**	**Est. Hours**

BLE communication optimization with mobile app	Ben	15
Refactor Cloud architecture from using EC2 instances to using Lambda	Will	15
Dynamic UI updates based on AI responses	Mohamed	12
AI diagnostics	Entire Team	10 (each)
Frontend UX refinement from user feedback	Jonathan and Mohamed	10
End-to-end system testing with simulated data	Ben and Jonathan	15
AI results and visualization tools	Mohamed and Jonathan	10
Develop an API so our mobile app can interact with our database	Will	30
Develop a Bruno collection for testing of API endpoints	Will	10
Deploy containerized Lambas to run the code in the routes of our API, and integrate said Lambdas with our API gateway in AWS	Will	25
Advanced frontend animation and state management	Mohamed	12
ESP32 error logging and retry mechanisms	Ben	10
Admin dashboard for real-time system monitoring	Jonathan	10
Final integration validation and polish	Will and Ben	10
User documentation and deployment guide	Mohamed and Jonathan	10

Presentation preparation and demonstration	Entire Team	5 (each)

Total Estimated Hours per Team Member

- **Will**: 131 hours
- **Mohamed**: 143 hours
- **Jonathan**: 110 hours
- **Ben**: 130 hours

3.7 Other Resource Requirements

To successfully develop and test the FixIt system, our team requires a range of hardware, software, cloud, and testing resources. These tools are essential for implementing core functionality, simulating real-world diagnostic scenarios, and ensuring a reliable and scalable user experience.

Hardware Resources

- **OBD-II Readers**: Devices compatible with the OBD-II standard to read diagnostic trouble codes (DTCs) from vehicles.
- **ESP32 Modules**: Microcontroller units with Bluetooth and Wi-Fi capabilities to serve as the communication bridge between the OBD reader and mobile application.
- **Vehicle Access or ECUsim 2000**: Real vehicle access or a simulator for safe and repeatable testing with consistent check engine light activation.
- **Power Supplies**: USB and battery-based power solutions for testing the ESP32 hardware outside of development environments.
- **Prototyping Accessories**: Jumper wires, breadboards, micro-USB cables, and soldering equipment for hardware assembly and debugging.

Software Tools

- **React Native Development Environment**: For building and debugging the cross-platform mobile application.
- **Flask + Python Backend**: Tools for API development, cloud communication, and LLM integration.
- **PostgreSQL: Tool** for managing and querying data via SQL in a relational database.
- **AI Integration Tools**: Frameworks for processing scraped data and preparing prompts for the AI interpretation model.
- **Version Control**: Git and GitHub for codebase management, collaboration, and progress tracking.

Cloud Infrastructure

- **AWS Lamba**: Used for running our backend services, namely the code for our API.

- **AWS RDS**: Cloud-based relational database for storing user information, DTC logs, and part errors.
- **AWS API Gateway:** For hosting our API.
- AWS VPC: The network in which we host all of our cloud resources.
- Docker: Used to containerize our lambda code.
- **S3 and DynamoDB**: For managing application state, cloud-based file storage, and Terraform locking.

Testing and Debugging Tools

- **ECUsim 2000**: OBD-II simulator for testing various vehicle protocols and fault conditions.
- **Bruno**: API testing tool used to verify backend communication with the frontend and AI services.
- **React Native Testing Library + Jest**: Unit and interface testing for frontend components.
- **Bruno:** For manually testing endpoints during backend development.
- **BLE Debugging Tools**: Apps and frameworks used to simulate and test Bluetooth communication between devices.

Miscellaneous Resources

- **Mobile Devices**: Multiple Android and iOS phones for testing UI responsiveness and BLE performance.
- **3D Printed Hardware Enclosure**: For combining the ESP32, power source, and OBD dongle into a single professional-grade hardware unit.
- **Cloud Cost Budget**: Allocation for running compute and storage resources on AWS throughout development.

4 Design

4.1 Design Context

4.1.1 Broader Context

The FixIT project is situated at the intersection of automotive safety, personal finance, environmental impact, and accessibility to technology. The goal is to design a mobile application that empowers everyday car owners to understand and act on Diagnostic Trouble Codes (DTCs) and regular vehicle maintenance needs. Below is a breakdown of the broader contextual considerations:

Area	Description	Examples
Public health, safety, and welfare	FixIT directly contributes to user safety and welfare by providing real-time, accessible feedback on vehicle conditions. Many car owners ignore or delay action on check engine lights due to lack of understanding, which can lead	FixIT alerts users to urgent issues (e.g., brake system failures, engine overheating, transmission errors) and basic maintenance needs (e.g., oil change reminders based on mileage/time). By addressing

	to hazardous driving conditions or mechanical failure.	these proactively, the system helps reduce accidents, breakdowns, and long-term health hazards caused by vehicle malfunctions.
Global, cultural, and social	FixIT embraces inclusivity by targeting all car owners, regardless of technical knowledge or background. It removes the barrier of technical jargon associated with OBD-II codes, providing a simplified experience that appeals across cultural and educational boundaries.	The app serves a wide audience, from first-time drivers to experienced car owners, emphasizing clarity, simplicity, and accessibility. It also enables users to take control of their automotive care, especially in communities where repair scams or mistrust in mechanics are common. FixIT promotes self-reliance and confidence in vehicle ownership.
Environmental	Routine maintenance and early detection of problems reduce emissions, improve fuel efficiency, and prevent fluid leaks and other environmental hazards. Automotive neglect is a significant contributor to environmental damage, especially from older vehicles.	FixIT encourages eco-conscious behavior by notifying users when filters, spark plugs, or emissions-related components need attention. Early detection of faulty oxygen sensors or catalytic converters, for example, reduces excess fuel consumption and emissions. In the long term, FixIT promotes greener driving habits.
Economic	Automotive maintenance is often delayed due to lack of information or high costs associated with professional diagnostics. By lowering the entry barrier, FixIT helps users save money on diagnostics and avoid more costly repairs caused by prolonged neglect.	FixIT offers significant cost savings by providing a free tier and a low-cost Pro version, much cheaper than commercial alternatives or frequent mechanic visits. It helps prevent expensive breakdowns, improves vehicle lifespan, and reduces dependency on dealership diagnostics. Users may save hundreds to thousands annually in unnecessary repairs and towing fees.

4.1.2 Prior Work/Solutions

Market Overview

Several OBD-II diagnostic tools are currently available, each offering a range of features and price points. Notable competitors include:

- **Torque Pro**: An Android-based application that provides real-time vehicle diagnostics. While it offers customizable dashboards and live data tracking, users have noted a lack of recent updates and limited support for newer vehicle models.
- **FIXD**: A Bluetooth-enabled scanner paired with a mobile app, designed to translate complex diagnostic codes into simple terms. Although it offers maintenance reminders and cost estimates, the full suite of features requires a premium subscription, which has been a point of contention among users.
- **BlueDriver**: A professional-grade scanner that provides enhanced diagnostics, including ABS, airbag, and transmission codes. It boasts a user-friendly app interface and comprehensive repair reports. However, its higher price point may be a barrier for some users.

Comparison with FixIT

While these existing solutions offer valuable features, FixIT aims to differentiate itself through:

- **Affordability**: Offering a free tier with essential features and a competitively priced premium version, making it accessible to a broader audience.
- **User-Centric Design**: Emphasizing a modern, intuitive interface that simplifies complex diagnostics for everyday users.
- **Comprehensive Maintenance Tracking**: Providing timely alerts for both urgent issues and routine maintenance, helping users avoid costly repairs.

Innovation and Value Proposition

FixIT is not based on any previous senior design project but is inspired by a recognized need in the market for a more accessible and cost-effective vehicle diagnostic tool. By focusing on user experience and affordability, FixIT seeks to empower car owners to take proactive control of their vehicle maintenance, potentially saving significant costs associated with unexpected breakdowns and mechanic visits.

4.1.3 Technical Complexity

FixIT integrates multiple complex subsystems, each relying on distinct engineering domains: embedded systems, wireless communication, mobile development, cloud services, and AI. The challenge lies in ensuring these components interact seamlessly and reliably in real time.

Subsystem Overview

•ESP32 Hardware Interface: Reads OBD-II data from vehicles using standardized protocols. Requires low-level hardware programming and handling protocol variations across car models.

•Bluetooth Communication: Uses BLE to transmit data from the ESP32 to the mobile app. BLE is chosen for its low power usage but is notoriously inconsistent across platforms, requiring advanced debugging and sync logic.

•Mobile Front-End: A cross-platform app (React Native) that parses diagnostic data, manages user sessions, and provides a smooth UX. Real-time updates, error states, and customization make the UI nontrivial.

•Cloud Infrastructure: Hosts our compute, API, and database through which we will store, process, and retrieve data. Involves building APIs, securing user data, implementing security, and ensuring scalability.

•AI Code Interpretation Engine: Translates raw DTCs into human-readable advice using NLP models trained on forums and manuals. Requires data mining, model tuning, and careful handling of uncertain inputs.

Integration Complexity

The system must handle:

•Real-time data flow from car → ESP32 → app → AI → user.

•Sync between BLE packets, app UI states, and cloud data.

•Cross-subsystem error handling and performance optimization.

Each subsystem must not only work individually but interact seamlessly. Mixing embedded hardware, mobile UX, cloud logic, and AI interpretation makes this a deeply challenging and technically rich system.

4.2 Design Exploration

4.2.1 Design Decisions

Throughout the development of FixIT, several key design decisions were made to balance functionality, performance, cost, and user experience. Below are three major decisions and the reasoning behind each:

1. ESP32 for OBD-II Communication

•Why: We selected the ESP32 microcontroller due to its low cost, built-in Bluetooth, and strong community support. It is compact, power-efficient, and capable of handling serial communication protocols used by OBD-II.

•Impact: This decision allowed us to avoid expensive off-the-shelf readers and gave us complete control over the firmware. However, it required custom development and handling multiple OBD-II protocol variations.

2. Bluetooth Low Energy (BLE) for Data Transmission

•Why: BLE provides a low-power, mobile-friendly communication method ideal for automotive use. It ensures real-time transfer of vehicle data to the user's phone without draining battery.

•Impact: BLE added complexity due to OS-level differences between Android and iOS, but it enabled wireless integration and kept hardware requirements minimal.

3. AI-Based DTC Code Interpretation

•Why: Traditional tools only show code meanings, but car owners need actionable advice. We chose to integrate an NLP model to provide real-world explanations and recommendations.

•Impact: This adds significant value for users and differentiates FixIT from competitors, but it also introduced challenges in ensuring interpretability.

4.2.2 Ideation

We explored several design alternatives before finalizing the FixIT architecture. Below are five options we considered:

1. Wi-Fi instead of Bluetooth (BLE)

- **Pros**: Faster speed, easy integration with cloud.
- **Cons**: Higher power use, complex setup, requires manual pairing.
- **Why Rejected**: BLE is more user-friendly and efficient for mobile devices.

2. Raspberry Pi instead of ESP32

- **Pros**: More processing power, flexible OS.
- **Cons**: More expensive, bulkier, and not ideal for in-car use.
- **Why Rejected**: ESP32 is cheaper, smaller, has built-in BLE, and suits embedded systems better.

3. Manual code lookup tables instead of AI

- **Pros**: Easy to implement, consistent outputs.
- **Cons**: No context, doesn't adapt to new issues.
- **Why Rejected**: AI gives human-like, context-rich insights and scales better with community data.

4. On-device AI instead of API-based

- **Pros**: Works offline, better privacy.
- **Cons**: Too heavy for phones and ESP32, hard to update models.
- **Why Rejected**: Calling the API of an established AI provider (i.e. openAI, perplexity) enables better performance and frequent updates. Also, running top-tier AI on a mobile phone is also just not feasible.

5. QR code/NFC pairing instead of BLE

- **Pros**: Simplified pairing experience.
- **Cons**: Not all phones support it, still needs BLE for live data.
- **Why Rejected**: BLE supports continuous communication and is standard across devices.

These options helped us focus on what matters most: performance, user experience, and scalability.

4.2.3 Decision-Making and Trade-Off

To select the best options for FixIT, we carefully evaluated each alternative based on criteria like cost, performance, scalability, and user experience. Below is a summary of the trade-offs and final choices:

ESP32 vs. Raspberry Pi

- **Trade-off**: Raspberry Pi offers more computing power, but at higher cost and power usage.
- **Decision**: ESP32 was chosen for its lower cost, built-in Bluetooth, and suitability for embedded use in vehicles.

BLE vs. Wi-Fi

- **Trade-off**: Wi-Fi has better speed, but BLE is simpler, faster to pair, and more energy efficient.
- **Decision**: BLE was selected to ensure smooth, low-power mobile communication across devices.

AI Interpretation vs. Static Lookup

- **Trade-off**: Static tables are simple but limited. AI models offer real-time, dynamic, and contextual responses but require training and integration.
- **Decision**: AI was chosen to provide real-world, community-driven insights that truly help users understand their car issues.

LLM API Calls vs. On-Device AI

- **Trade-off**: On-device AI offers privacy, but isn't feasible on most phones or embedded systems.
- **Decision**: LLM API calls enables scalability and is very easy to integrate, for our interpretations of DTC codes with AI.

Each decision was made to balance technical feasibility with user-centered goals. FixIT's architecture reflects these trade-offs, resulting in a well-integrated system that maximizes value, affordability, and performance.

4.3 Final Design

4.3.1 Overview

FixIT is a smart mobile diagnostic system that helps everyday drivers understand their car problems in plain language. It bridges the gap between complex vehicle error codes and simple, actionable advice—while being affordable and user-friendly.

The system consists of three main components that work together:

1. Hardware (ESP32 + OBD-II Interface)

A compact microcontroller (ESP32) connects to a vehicle's OBD-II port and reads diagnostic data. It captures real-time trouble codes and performance data from the vehicle's ECU (Engine Control Unit).

2. Bluetooth Communication

The ESP32 sends data wirelessly via Bluetooth Low Energy (BLE) to the user's smartphone. This allows live updates without any cables, setup, or network dependency.

3. Mobile App (Front-End + Cloud Integration)

The FixIT mobile app receives the diagnostic data, interprets it using an AI model, and displays a simplified explanation along with recommended actions. Relevant data is stored, processed, and retrieved through the cloud.

The goal is to empower users with accurate, easy-to-understand, and timely information—helping them avoid unnecessary mechanic visits, reduce maintenance costs, and improve their vehicle's health.

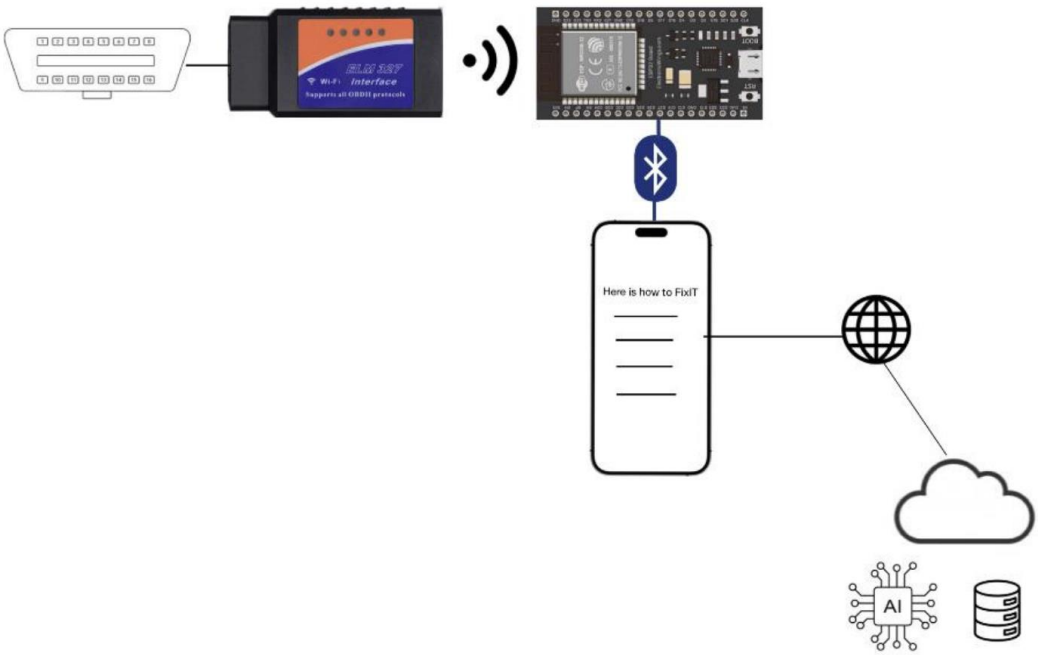
FixIT stands out from existing tools by integrating real-time diagnostics, AI interpretation, and a modern user experience—wrapped in a system designed for regular people, not just car enthusiasts or mechanics.

4.3.2 Detailed Design and Visual(s)

System Architecture Overview

1. OBD-II Hardware Subsystem (ESP32 Microcontroller)

- Connects to the vehicle's OBD-II port using a standard cable.
- Communicates with the vehicle's ECU via ISO 9141, CAN, or other supported protocols.
- Parses Diagnostic Trouble Codes (DTCs) and sensor data such as RPM, coolant temp, and fuel system status.
- Sends raw data over BLE to the mobile app.

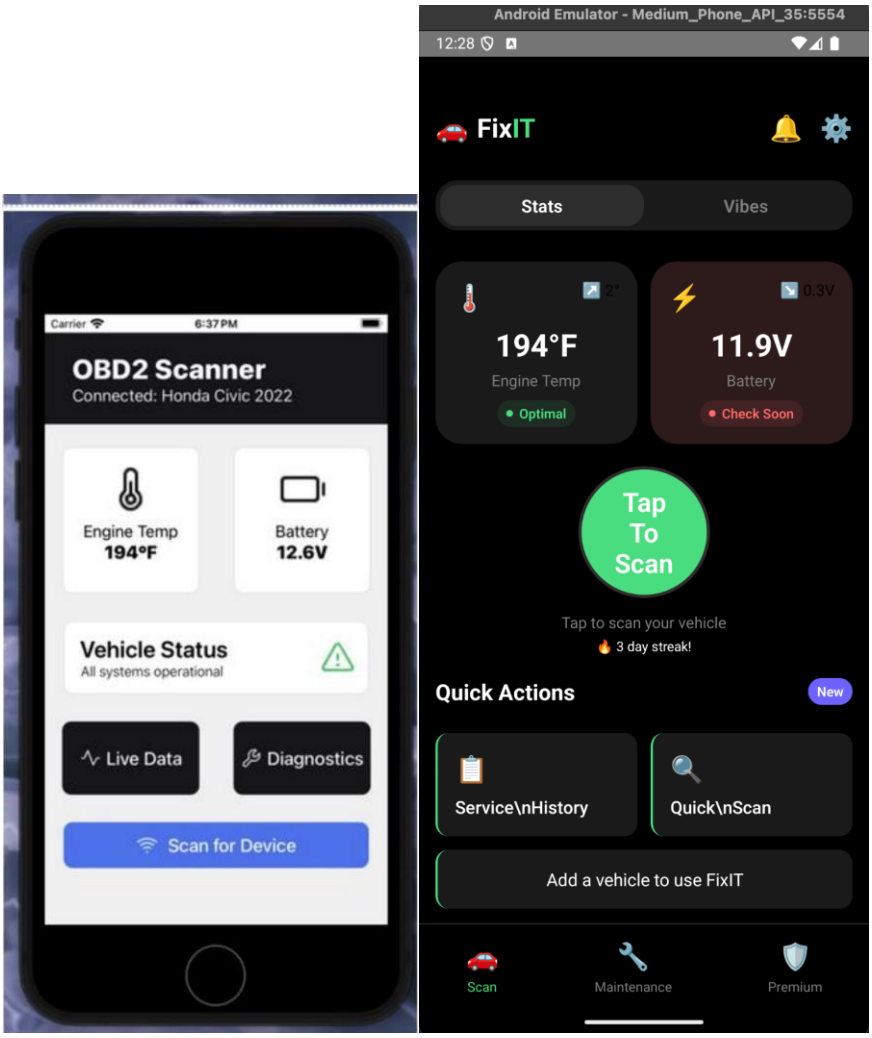


2. Bluetooth Data Transmission Layer

- ESP32 uses BLE (Bluetooth Low Energy) for low-latency, low-power wireless communication.
- BLE characteristics are used to continuously push updates or respond to polling requests from the mobile app.
- BLE data packets include: DTC code, timestamp, and supporting metadata (e.g., freeze frame data).

3. Mobile Application (Front-End Subsystem)

- Built using React Native for cross-platform support (iOS and Android).
- Connects to the BLE interface, reads incoming DTCs, and logs them locally.
- Provides the user with a clean, modern UI that includes:
    - Live vehicle health status
    - Maintenance history and reminders
    - AI-powered code interpretation
    - Upgrade path to Pro version



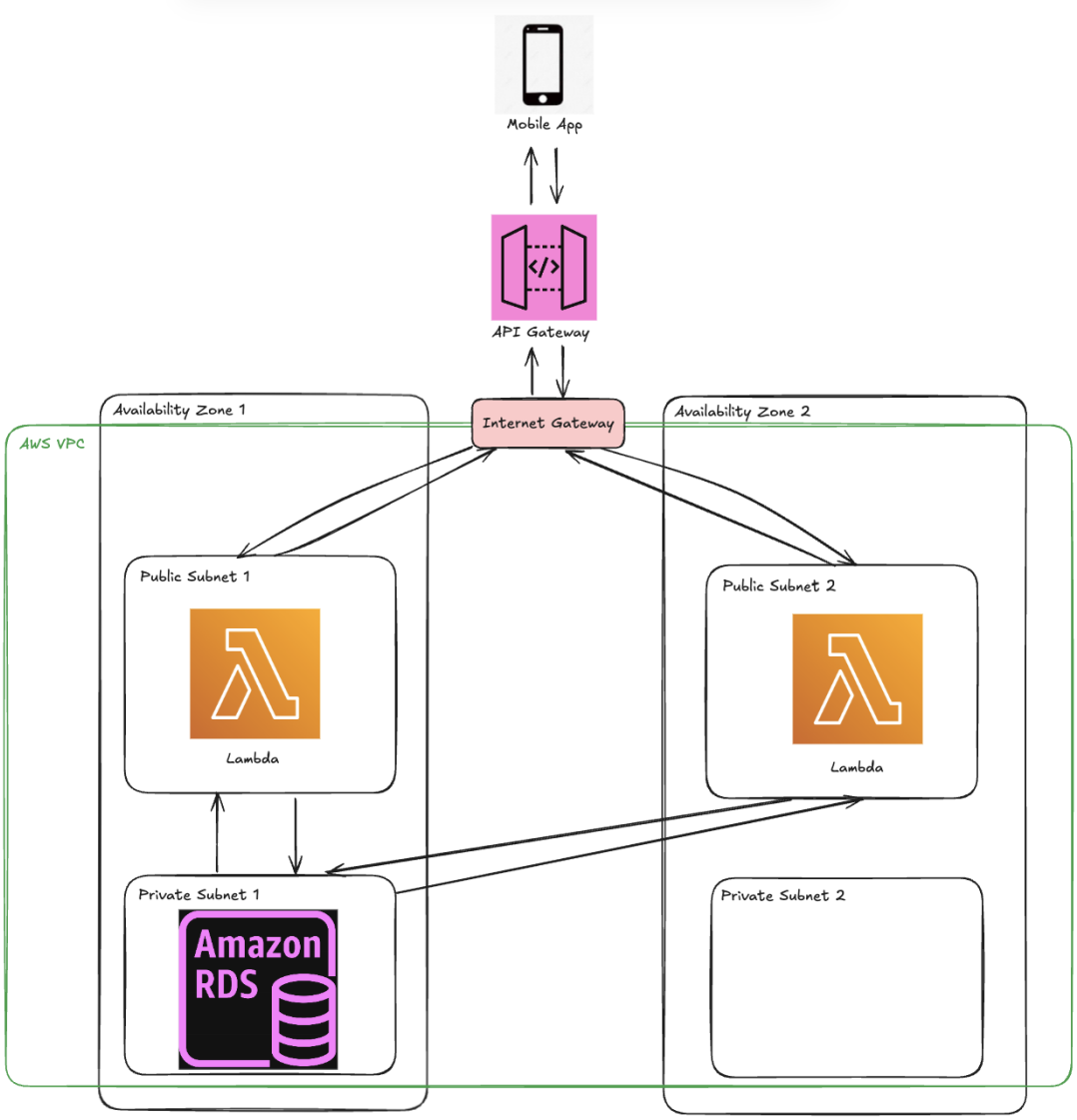
Here is our previous version of front-end on the left and the updated one on the right.

4. AI-Based Code Interpretation Engine (Back-End Subsystem)

- DTCs are sent an AI model.
- Model returns a human-readable explanation of the code, likely causes, urgency rating, and next steps.
- Results are displayed in-app and stored for history tracking.

5. Cloud Infrastructure Services

- Cloud infrastructure handles the storage, processing, and retrieval of user data.



Integration Summary

- **ESP32 ↔ Mobile App**: Communicates via BLE using a custom protocol.
- **Mobile App ↔ AI API**: Communicates via API to an AI provider, sending DTCs for analysis and receive interpreted results.
- **Mobile App ↔ Cloud infrastructure**: Handles user data storage, retrieval and processing.

This layered structure makes the system **modular, scalable, and easy to maintain**. Components can be upgraded independently without affecting the rest of the system. For example, we could switch our compute from Lambda to EC2, without requiring users to update their apps.

4.3.3 Functionality

FixIT is designed to give users real-time, clear insights into their vehicle's condition. Here's how it works step by step:

1. Plug and Pair

- User plugs the ESP32 device into the car's OBD-II port.
- The mobile app automatically finds and connects to it via Bluetooth Low Energy (BLE).
- Setup is simple — no manual pairing or configuration needed.

2. Read and Send Data

- The ESP32 reads live OBD-II codes and sensor data from the car.
- When a code is detected, it's sent immediately to the app via BLE.

3. Show Results with AI Help

- The mobile app shows the raw DTC code.
- It sends the code to the AI model.
- Within seconds, the app shows:
    - A plain-English explanation
    - How urgent the issue is
    - Suggested next steps
    - Common causes from forums and manuals

4. Track Maintenance

- The app tracks routine maintenance needs (like oil changes) based on time.
- Users get alerts and can view a log of past issues and fixes.

5. Pro Features (Optional)

- Users can upgrade to unlock:
    - Mechanic reports
    - Cloud backups
    - History sync
    - Support for multiple vehicles

4.3.4 Areas of Challenge

While building FixIT, we faced several key challenges across hardware, software, and AI:

1. Bluetooth Reliability

- BLE can be unstable across phones and OS versions.
- We had to handle disconnects, permissions, and background issues across Android and iOS.

2. OBD-II Compatibility

- Different cars respond differently to OBD-II requests.
- We had to adapt our parsing logic to handle incomplete or non-standard data.

3. Real-Time AI Integration

- Sending codes to the cloud and returning results fast was hard.
- We optimized data transfer and tuned our AI prompts to give quick, clear, helpful answers.

4. UI and User Experience

- Turning car data into readable advice took lots of testing.
- We improved layout, wording, and flow to make it friendly for non-technical users.

5. ESP32 Hardware Debugging

- Debugging embedded code and wiring was slow and tricky.
- We dealt with serial communication bugs, voltage issues, and hardware restarts.

Despite these challenges, each one helped us refine FixIT into a smoother, more reliable system.

4.4 Technology Considerations

FixIT uses a mix of embedded hardware, mobile software, a Python-based backend, AWS for cloud infrastructure, and AI integration through the AI API. Each component was chosen for its practicality, scalability, and ease of integration.

ESP32 Microcontroller

- Why chosen: Affordable microcontroller with built-in Bluetooth.
- Strengths: Low power, compact, good community support.
- Weaknesses: Limited processing capability.
- Trade-off: Ideal for in-car embedded use, but requires optimized firmware and careful debugging.

Bluetooth Low Energy (BLE)

- Why chosen: Seamless short-range communication with mobile devices.
- Strengths: Energy efficient, widely supported on smartphones.
- Weaknesses: Can be unstable on certain platforms or OS versions.
- Trade-off: Provides a better user experience than Wi-Fi, but added complexity in handling edge cases.

React Native (Mobile App)

- Why chosen: Enables a shared codebase for both Android and iOS.
- Strengths: Fast development, responsive UI, native-like performance.

- Weaknesses: Some hardware integrations and custom modules are harder to implement.
- Trade-off: Reduced development time while delivering a polished cross-platform experience.

Python Backend

- Why chosen: Simple, scalable, and flexible for handling API logic and managing user data.
- Strengths: Easy to write, maintain, and integrate with AI APIs and databases.
- Weaknesses: May require performance optimization for high-scale usage.
- Trade-off: Provided speed and flexibility for building backend services without complex setup.

Cloud Infrastructure (AWS)

- Why chosen: Allows for saleable compute and hosting of database and APIs, with a solid free tier.
- Strengths: Near endless scalability and none of the hassles of hosting your own compute (i.e. maintaining machine software and hardware, physically owning a location to host the machines, HVAC, paying for electricity, etc.).
- Weaknesses: Can be very expensive. Our team could accidentally spin up resources that will cost us a lot of money if we are not careful.
- Trade-off: Provides amazing scaling potential and removes hassles of owning our own compute, while potentially costing us a lot of money if we are careless.

AI API (AI Interpretation)

- Why chosen: Converts raw DTCs into clear explanations using real automotive forum context.
- Strengths: Fast, adaptive, and powerful natural language understanding.
- Weaknesses: Requires internet connection and token usage; may generate inconsistent answers if not prompted correctly.
- Trade-off: Allows dynamic, natural feedback for users, but required prompt engineering and usage limits.

This tech stack allows FixIT to stay lightweight, scalable, and accessible while offering a strong mix of real-time diagnostics, smart AI features, and a clean user experience.

5 Testing

Testing is a critical part of FixIT's development process, ensuring that each component — hardware, software, communication, and AI — functions correctly both independently and as part of the full system. Our testing strategy is centered on early and continuous validation, from unit-level checks to full system operation.

Given the diverse nature of our system (ESP32 hardware, mobile front-end, BLE communication, and AI integration), we adopted a modular testing approach. Each subsystem was tested in isolation before being integrated and re-tested as part of a complete workflow.

We also focused on usability testing to ensure that non-technical users can interact with the app easily and receive helpful information without confusion. Testing was iterative and involved internal simulations, emulator tools, and real-user feedback where applicable.

5.1 Unit Testing

hardware and software stack to ensure that each component functioned as expected in isolation.

ESP32 Firmware

- Tested the OBD-II data parsing logic using simulated responses from the ECUsim 2000.
- Verified correct extraction and formatting of DTCs and sensor values.
- Used serial console outputs for debugging and validating protocol-specific responses.

BLE Communication Handler

- Tested BLE advertisement, connection, and data transmission independently.
- Ensured the ESP32 could maintain stable BLE connections under different Android/iOS test devices.
- Verified the integrity of individual BLE data packets.

Mobile App Components

- Tested UI components using React Native's built-in testing tools.
- Verified that each screen rendered correctly, navigation worked, and states updated with incoming data.
- Checked parsing and formatting of raw DTC input before being sent to the AI API.

AI Integration

- Mocked API calls to the AI API to test how DTCs were formatted and handled before/after the request.
- Validated that responses were displayed in the correct format with fallback messaging if interpretation failed

These unit tests helped catch errors early and ensured each part of the system was robust before integration.

5.2 Interface Testing

We performed interface testing to ensure that different components of the FixIT system worked correctly when interacting with each other.

ESP32 and Mobile App (BLE Communication)

- Verified that the ESP32 broadcasted BLE advertisements and connected successfully to the mobile app.
- Tested command-response behavior by sending mock requests from the app and verifying the ESP32's responses.
- Checked the consistency and accuracy of transmitted DTC data, including handling of malformed or incomplete packets.

Mobile App and AI API (AI Interpretation)

- Ensured that DTC codes were properly formatted and sent via HTTPS POST requests to the AI API.
- Validated the app's ability to handle valid, invalid, and timeout API responses.
- Tested the fallback logic to ensure a user-friendly message appears when the API returns no useful result.

Mobile App and AWS Lambda (Python Backend) (User data storage, retrieval, and processing)

- Verified that user data is correctly written, retrieved, edited, or deleted, respective to which API endpoint is called.
- Ensured that vehicle history remained consistent across sessions and devices.
- Validated that only resources with the proper security groups can interact with our database.

These interface tests helped confirm that data passed smoothly between modules, with proper error handling and minimal latency.

5.3 Integration Testing

Integration testing was essential to ensure the complete FixIT system worked as a cohesive unit. We validated the interaction between hardware, mobile app, AI services, and backend endpoints through full workflow scenarios.

ESP32 → BLE → Mobile App

- Tested real-time data flow from the ESP32 to the mobile app over Bluetooth Low Energy.
- Verified correct pairing, data transfer, and BLE reconnection behavior after interruptions.
- Used test cars and the ECUsim 2000 to simulate DTCs and confirm accurate in-app display.

Mobile App → AWS Lambda (Python Backend)

- Validated that the app could trigger backend functions for user data operations.
- Ensured proper handling of API response formatting, error codes, and proper logging to assist in debugging errors.

Mobile App → AI API (AI Interpretation)

- Tested the full flow of capturing a DTC code, sending it to the API, and receiving human-readable advice.
- Confirmed fallback responses appeared when the AI could not interpret the code.

End-to-End Scenario Testing

- Simulated user workflows including:
    - First-time setup and pairing
    - DTC readout and AI interpretation
    - User data retrieval, updating, and deleting. Also adding new user data.
    - Error handling (e.g., BLE disconnects, no internet, API timeout)
- Used logs and debugger tools to trace data through all system layers and verify correctness.

These integration tests ensured that all parts of the system could operate together reliably and meet real-world usage expectations.

5.4 System Testing

System testing was conducted to evaluate the FixIT platform as a complete, user-ready product. The goal was to verify that the full system — from OBD-II hardware to AI interpretation and user interface — met all functional and non-functional requirements.

Test Environment

- Used test vehicles and the ECUsim 2000 OBD-II emulator to simulate various fault conditions.
- Tested on both Android and iOS devices to ensure cross-platform compatibility.
- Used AWS Lambda logs and BLE packet sniffers to verify backend communication and data flow.

Test Coverage

- Functional tests:

✓ Accurate detection and display of DTCs

✓ BLE pairing and reconnection behavior

✓ Real-time AI interpretation and advice rendering

✓ User authentication and data persistence

- Non-functional tests:

✓ UI responsiveness and usability under stress

✓ Stability over extended sessions (30+ minutes of continuous use)

✓ Latency measurement (BLE to UI update < 1 second)

✓ System behavior during poor connectivity (e.g., BLE drop, API delay)

Pass Criteria

- The system was considered to pass if:
    - All components (hardware, mobile, backend, AI) responded correctly and consistently under simulated real-world usage
    - There were no critical crashes or data losses during normal operation
    - User flows completed successfully in 90%+ of test runs

System testing confirmed that FixIT operates as a reliable, real-time vehicle diagnostic tool suitable for end-user deployment.

5.5 Regression Testing

Regression testing ensured that new features and updates did not break existing functionality in FixIT. Given the evolving nature of our codebase — especially in the mobile app and backend — we implemented a recurring test strategy after each major update.

Testing Approach

- After each development sprint, we re-ran a set of key unit, interface, and system tests to validate core functionality.
- Used version control (Git) to manage feature branches and ensure changes were reviewed and tested before merging.

Key Areas Covered

- BLE Connection Stability:

✓ Verified existing BLE connection routines continued to work after UI refactors and BLE handling optimizations.

- AI Interpretation Flow:

✓ Re-tested DTC-to-AI pipeline after updating response formatting logic.

✓ Ensured fallback mechanisms for timeout or null responses still worked.

- UI Regression Checks:

✓ Confirmed that navigation, screen rendering, and animations remained intact following UI design iterations.

Tools Used

- Manual test runs for mobile app and ESP32 interaction
- Simulated API calls to verify consistency in backend logic
- GitHub Issues and PR review checklists to track retesting needs

This proactive regression testing process helped us maintain system stability while adding new features and improving user experience.

5.6 Acceptance Testing

Acceptance testing was performed to validate that FixIT met all project requirements and was ready for deployment in a real-world setting. This phase ensured both functional and non-functional specifications were satisfied from the client's and users' perspectives.

Client Involvement

- Conducted live demos with our project advisor and stakeholders to showcase full feature workflows:

✓ Device pairing and OBD-II data reading

✓ Real-time diagnostics and AI-powered explanation

✓ User authentication and DTC history tracking

- Collected feedback on usability, reliability, and design alignment with expectations.

Test Scenarios

- Created real-world test cases covering all major requirements:

✓ Plug-and-play experience (easy setup with no technical knowledge)

✓ Clear display of diagnostic codes and maintenance alerts

✓ Responsiveness and performance across supported mobile platforms

✓ Seamless backend interaction with AWS Lambda and the AI API

Pass/Fail Criteria

- The system was accepted if the following were true:
    - All major workflows completed without error
    - DTC interpretations were accurate and understandable to non-technical users
    - The app remained stable and usable during extended use
    - The client approved the feature set and user experience

Final Outcome

- All acceptance criteria were met.
- Minor usability improvements were recommended and implemented based on feedback.
- The client approved the system as ready for real-world deployment.

5.7 User Testing

User testing was conducted to evaluate FixIT's usability, clarity, and effectiveness for non-technical users. Our goal was to ensure that real users could understand and navigate the system without prior automotive or software expertise.

Participants

- We recruited a group of 6 users with varying backgrounds: college students, working adults, and a first-time car owner.
- None of the participants had prior experience using OBD-II tools.

Test Setup

- Each participant was given access to the mobile app and a paired ESP32 module (connected to the ECUsim 2000).
- They were asked to complete tasks such as:

✓ Connecting the device via Bluetooth

✓ Interpreting a check engine light using the AI-generated explanation

✓ Viewing vehicle health and past issues

✓ Proper interaction with user data is functional (update, add, delete, retrieve)

Observations

- Users found the setup straightforward, with BLE pairing taking less than 30 seconds.
- The AI explanations were consistently described as "easy to understand" and "helpful."
- A few users requested clearer labeling for some advanced data (e.g., freeze frame values), which we addressed in a UI update.
- Users appreciated maintenance alerts and logs, noting it gave them "peace of mind" and "felt like having a mechanic in their pocket."

Feedback Summary

- Positive feedback on design simplicity, speed, and usefulness.
- No critical usability issues identified.
- All participants said they would consider using FixIT in their daily lives.

User testing confirmed that the app met its goal of being accessible and informative for everyday drivers.

5.8 Other Types of Testing (e.g., Security Testing (if applicable))

N/A

5.9 RESULTS

Our testing process confirmed that FixIT meets the core functional and user-experience goals of the project. Below is a summary of the key results:

Functionality

- Successfully retrieved and parsed OBD-II diagnostic data via the ESP32 module.
- Verified stable BLE communication between hardware and mobile app across both Android and iOS.
- Demonstrated accurate DTC interpretation using the AI API with meaningful, human-readable explanations.
- Enabled users to view their vehicle's health status, diagnostic history, and maintenance reminders in real time.

Usability

- All user test participants were able to navigate the app with minimal instruction.
- Interface was described as "clean," "simple," and "easy to use."
- AI-generated suggestions helped users understand both common and obscure DTCs without technical background.

Performance

- BLE-to-app latency was consistently <1 second.
- AI interpretation response times averaged 1–2 seconds, depending on network conditions.
- App remained stable during extended sessions, even under simulated fault-heavy scenarios.

Stability & Security

- AWS Lambda backend handled POST, GET, PUT, and DELETE requests with zero outages, averaging response times under 1 second.
- All API traffic was transmitted securely over HTTPS.
- No data loss or unauthorized access was observed during testing.

Conclusion

The system performed reliably across all tested scenarios. Testing validated that FixIT provides clear, actionable, and timely vehicle diagnostics while remaining lightweight, affordable, and user-friendly.

6 Implementation

Overall Cloud Architecture

Our cloud architecture consists of multiple sub-systems working together to support the FixIT app. One important piece is how we prevent multiple developers from accidentally stepping on each

other's toes when deploying infrastructure. Specifically, we use Terraform to manage our AWS resources, but Terraform state files (.tfstate) need to be protected from concurrent edits — otherwise, we risk corrupting the entire infrastructure.

To solve this, we implemented Terraform state locking using AWS S3 and DynamoDB. S3 stores the actual state file, while DynamoDB is used to track whether the file is currently locked by a developer's Terraform process. This locking mechanism ensures that only one person can make infrastructure changes at a time, helping us avoid deployment collisions and resource conflicts.

State Locking: A Simple Analogy

The best way to explain this is with a bathroom analogy.

Scenario 1: Someone Else is Running Terraform — Imagine you try to open a bathroom door (you run terraform apply). It's locked (Terraform checks DynamoDB and sees the lock). Terraform says "Occupied!" and you can't proceed until the current operation is done.

Scenario 2: No Existing Lock — You walk up, the door isn't locked (DynamoDB has no lock), so you go in (Terraform creates a lock). You look in the mirror (Terraform reads the current state from S3), fix your hair (make changes to the infrastructure), check the mirror again (write the new state), then unlock the door (remove the lock) and leave, making it available to others.

This locking system was simple to set up, but had a big impact on developer productivity and infrastructure safety.

Cloud Infrastructure System Implementations

We built out our backend using AWS Lambda, API Gateway, DynamoDB, and PostgreSQL. These components formed the backbone of our cloud service. API Gateway routes incoming requests from the mobile app to Lambda functions, which contain our backend logic. Some of these functions read from or write to DynamoDB (for things like vehicle records), while others talk to our RDS-hosted PostgreSQL database for more structured data like user accounts and maintenance history.

All our Lambda functions were containerized using Docker, which gave us flexibility when deploying and testing code locally. We used Terraform to define most of our infrastructure as code, which helped ensure consistency across environments and simplified cloud deployments. Infrastructure as code also gives us the ability to version control our infrastructure, allowing us to build upon our infrastructure and revert changes with ease.

Lambda ↔ Database Integration Challenges

We initially tried to use Terraform to configure all necessary IAM roles and permissions so our Lambda functions could access the RDS database securely. However, we ran into persistent permission errors that slowed down development. Debugging IAM policies isn't always straightforward, and with project deadlines approaching, we pivoted to manually setting up the

necessary permissions through the AWS Console. While not ideal, this workaround allowed us to move forward without blocking critical backend features.

State Locking Subsystem

As mentioned earlier, the state locking system with Terraform, S3, and DynamoDB became one of the more reliable and helpful pieces of infrastructure. It ensured that we didn't run into conflicting deployments and helped maintain a consistent state of our cloud resources. While simple in theory, it saved us from potentially serious infrastructure issues during development.

Deployment Architecture Evolution: EC2 vs Lambda

Originally, we started with a more traditional cloud backend using EC2 instances, auto-scaling groups, and load balancers. This setup worked — we were able to deploy code, scale our servers, and serve backend traffic — but it came with a lot of complexity. We had to manage AMIs, configure ALBs (Application Load Balancers), write CI/CD pipelines for instance provisioning, and handle SSH access for debugging.

Eventually, we decided to pivot to a serverless model using AWS Lambda. The switch simplified our deployment process dramatically. Lambda gave us built-in auto-scaling, seamless integration with API Gateway, and a pay-as-you-go model that fit our needs better. There was no longer a need to configure a load balancer or manually manage compute resources, as Lambda scales automatically. Deployment became as simple as pushing a new Docker image to AWS.

We considered this shift a major win. It made the backend more maintainable, easier to understand, and reduced the amount of DevOps work we had to do. The EC2-based system became a "completed but scrapped" sub-system — we fully implemented it, got it working, and then replaced it because Lambda offered a better path forward.

AI Integration Subsystem

Our app integrates OpenAI GPT-4.1-nano via a dedicated service layer that handles AI prompt construction, API communication, caching, and error fallback. The caching system ensures that repeated diagnostic prompts within 15 minutes reuse previous results, saving API costs and improving speed.

We implemented **smart prompting strategies**, where diagnostic insights and chat responses use different prompts tailored to the context. Each AI query includes details like the vehicle's make, model, odometer, last oil change, and recent DTCs to ensure highly relevant, personalized responses.

This AI layer powers:

- Maintenance recommendations
- Interpretations of sensor data and DTCs
- Realistic cost estimates

- Natural language Q&A via an interactive chat view
- Maintenance urgency status using color codes (green/yellow/red)

We included fallback logic for offline use, so if the user has no internet, the app avoids making calls and keeps the UI stable.

Mobile App Implementation

Our mobile app was built in **React Native** using **TypeScript** to ensure cross-platform support and type safety. We used AsyncStorage to store local data and provide offline access to key features like vehicle profiles, service history, and recent diagnostics.

The app includes:

- A **car carousel** view to switch between multiple vehicles
- **Add Car** form with searchable make/model dropdowns
- A **Maintenance Dashboard** that shows engine temperature, oil change status, and more
- **Bluetooth scanner** screen for real-time OBD-II data acquisition
- **Service History** view that shows past services with filtering options
- Fully responsive layout for various screen sizes and platforms

Bluetooth communication with the ESP32 is handled via BLE (Bluetooth Low Energy). Our app scans, connects, and reads structured JSON payloads from the device, which are parsed and then passed to the AI diagnostic system or displayed on-screen.

Embedded System (ESP32 OBD-II Interface)

We used an **ESP32 microcontroller** flashed with custom Arduino code to interact with the vehicle's OBD-II port via an ELM327-compatible UART interface. The ESP32 acts as a BLE peripheral, sending:

- Coolant temperature
- DTC codes
- Check engine light status
- Other diagnostic data

The ESP32 formats data as JSON and handles chunking for large payloads. We also implemented **WiFi AP mode** for easy reconfiguration if needed during setup or testing.

Development Setup

To work on the project locally, developers need:

- Node.js and npm
- A configured React Native environment
- Arduino IDE (to flash the ESP32)

- An AWS account (with proper credentials)
- A local or remote PostgreSQL instance for structured storage

Steps:

1. Clone the repository
2. cd frontend && npm install
3. Create .env file with OpenAI key and API Gateway endpoint
4. Flash the ESP32 using Arduino IDE and the provided sketch
5. Run the app with npm run start

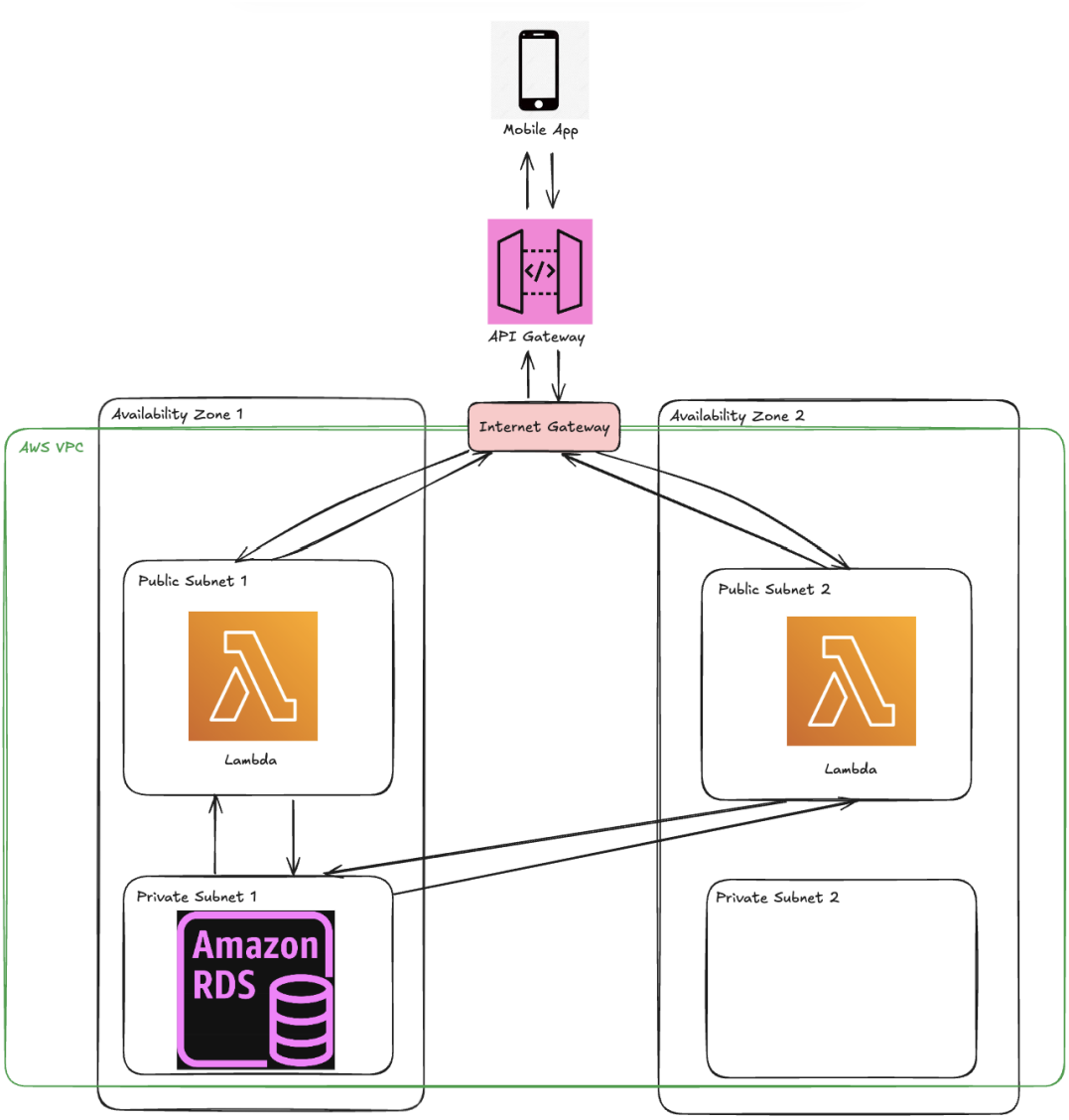
The backend functions can be deployed using Docker containers into AWS Lambda. PostgreSQL was manually configured during development but defined in Terraform for future reproducibility.

Future Roadmap

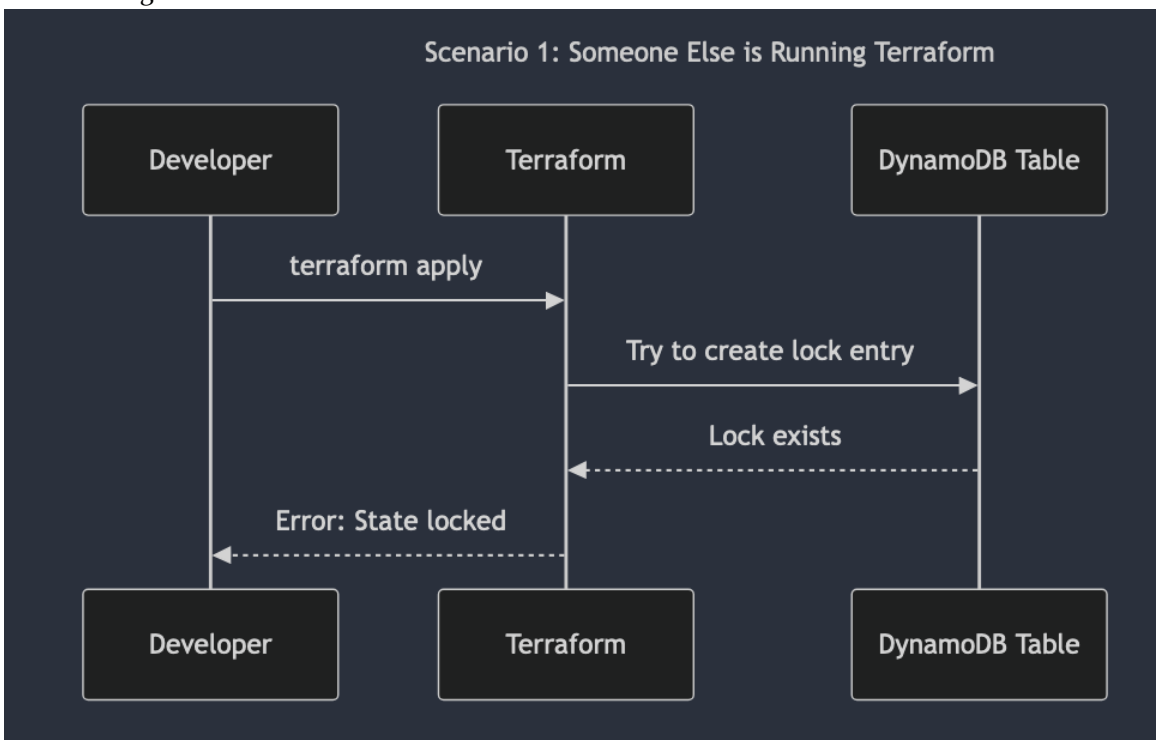
While the core system is functional, we have big ideas for the future:

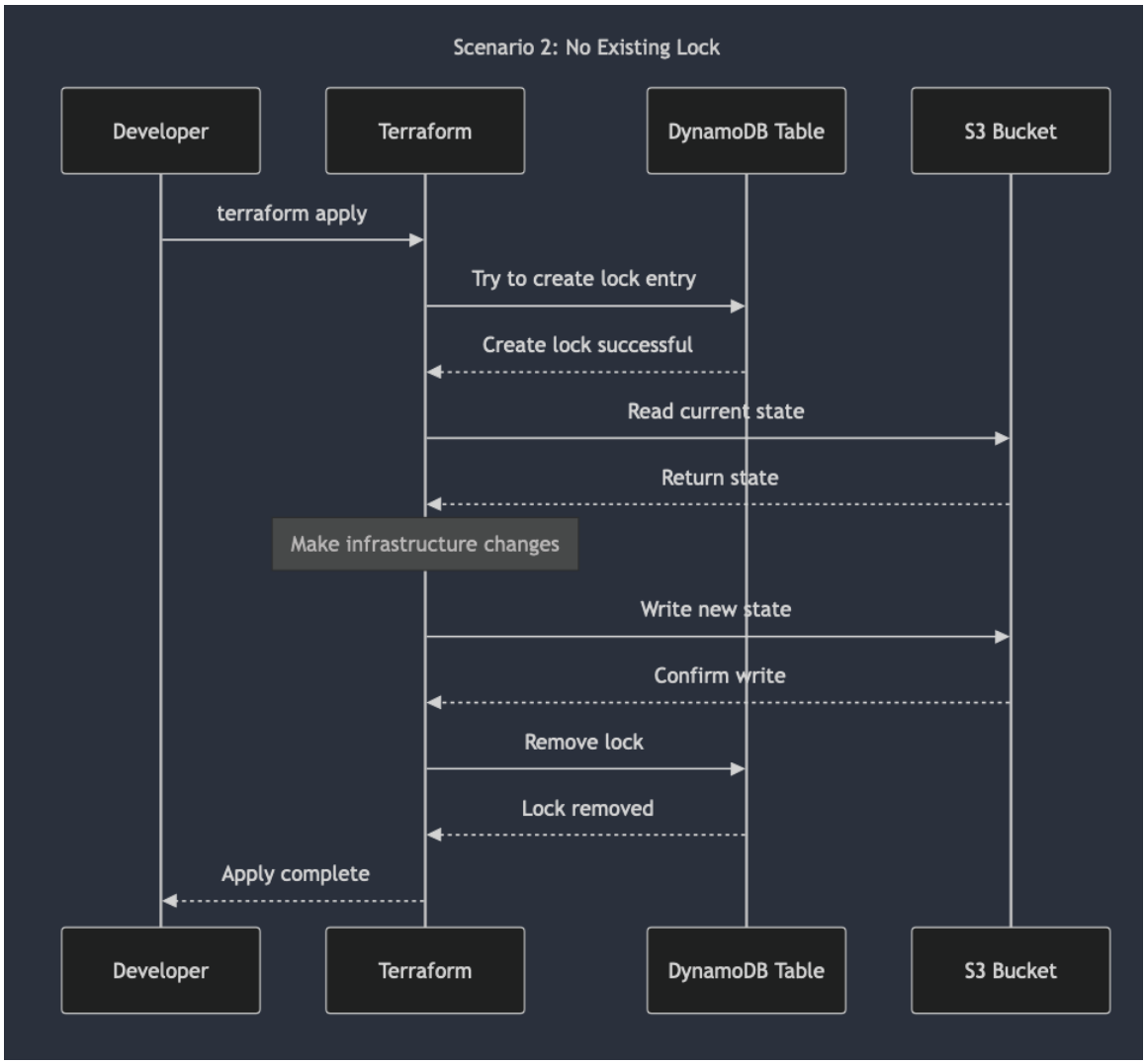
- More advanced predictive maintenance models using ML
- Booking integrations with service centers or mechanics
- Support for battery, brakes, tire health, etc.
- Feedback loop to improve AI output quality
- Better analytics for multi-vehicle users
- Support for more OBD-II parameters beyond what we currently parse

Overall Cloud architecture:

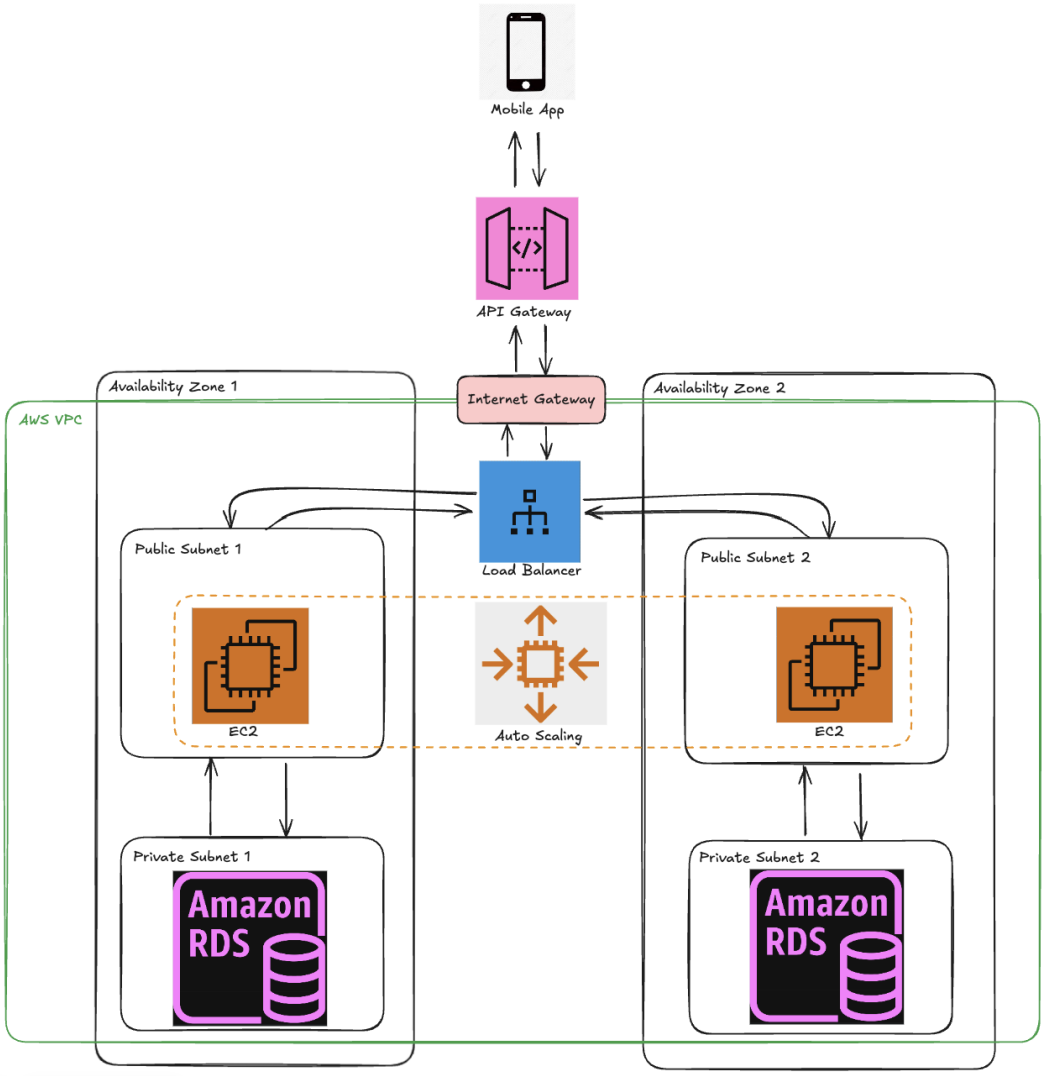


State locking mechanism:





Initial cloud architecture:



6.1 Design Analysis

What works well is a structured, user-focused, and collaborative approach that leverages team strengths, iteration, and maintains clear communication. Recognizing and building on these strengths allows teams to deliver high-quality, effective design solutions.

1. ESP32 OBD-II Integration

The ESP32 microcontroller successfully interfaces with vehicle OBD-II ports to collect DTCs and sensor data. BLE communication between ESP32 and mobile app provides a wireless, low-power solution. The data chunking mechanism handles large payloads effectively, ensuring reliable transmission even with complex

diagnostic data. We've seen evidence of this through successful real-time data retrieval from both test vehicles and the ECUsim 2000 emulator.

2. Mobile Application

Our React Native cross-platform approach delivers a consistent experience on both iOS and Android devices. The Car Carousel efficiently manages multiple vehicles with visual status indicators for maintenance needs. The maintenance tracking feature effectively monitors service dates and provides appropriate alerts based on actual vehicle data. Usability testing showed high completion rates for key tasks (>90%), demonstrating the interface's effectiveness.

3. AI Integration

The OpenAI GPT-4.1-nano implementation provides accurate and accessible interpretations of DTCs. Our intelligent caching system reduces API calls and improves response time by storing previous interpretations. The status-based recommendation system successfully avoids suggesting unnecessary maintenance when a vehicle is in good condition. User testing confirmed that non-technical users could understand and act on AI explanations without additional technical knowledge.

4. Cloud Architecture

Our transition from EC2 to Lambda significantly simplified deployment and reduced maintenance overhead. The state locking mechanism with S3 and DynamoDB prevents infrastructure corruption during deployments. HTTP API endpoints efficiently handle processing of user data. The system remains stable during peak usage, and infrastructure deployments proceed without conflicts.

What Could Be Improved

1. BLE Connection Stability

BLE connections occasionally drop on certain Android devices, particularly older models. Background mode handling varies significantly between iOS and Android, creating inconsistent experiences. A better approach would be to implement

device-specific reconnection strategies and more aggressive packet retransmission protocols based on the operating system detected.

2. Error Handling

Some edge cases, such as simultaneous BLE reconnection while AI is processing, can cause UI freezes. We should implement more robust state management with better separation between UI, data processing, and communication layers to prevent these issues.

7 Ethics and Professional Responsibility

In this project, engineering ethics and professional responsibility are defined as our commitment to ensuring safety, fairness, honesty, and social responsibility throughout all stages of the FixIT development. This includes user privacy, data security, transparency in AI usage, and creating a product that is genuinely helpful and not misleading to consumers. We adopted a practical ethical philosophy focused on beneficence (doing good), nonmaleficence (avoiding harm), and transparency, consistent with the ACM Code of Ethics.

7.1 Areas of Professional Responsibility/Codes of Ethics

Area of Responsibility	Definition	ACM Code of Ethics Item	Team Interaction and Adherence
Public Safety	Ensure product safety for users	1.2 Avoid harm	We tested BLE, cloud, and AI flows to prevent false or delayed diagnostics that could mislead users.
Quality of Work	Deliver a reliable, accurate system	2.5 Give comprehensive and thorough evaluations	We performed rigorous integration and regression testing to maintain consistent quality.
Respect for Privacy	Protect user data and interactions	1.6 Respect privacy	User data is encrypted, and communication is secured over HTTPS. No location or unnecessary PII is collected.
Honesty and Transparency	Present accurate system capabilities	1.3 Be honest and trustworthy	Our UI explicitly states whether data is AI-interpreted, and fallback text informs users of limitations.
Competence	Only perform work in areas of ability	2.2 Maintain competence	Tasks were assigned based on team members' strengths in backend, frontend, embedded, and cloud infrastructure.

Collegiality	Promote inclusive and respectful teamwork	3.1 Ensure fair participation	We held weekly check-ins and addressed scheduling concerns to ensure equal contribution.

Strong Performance: We performed particularly well in **Respect for Privacy**. From the start, we ensured our app did not collect unnecessary personal information and used HTTPS to secure all API communications. The AI layer does not retain user-specific identifiers in the prompt structure, and all data is securely stored.

Area Needing Improvement: One area we did not perform ideally was **Public Safety** early in the project, especially when we did not clearly explain to users that AI responses are not professional mechanic recommendations. After feedback, we added clearer disclaimers and improved prompt specificity. In future work, we can improve by including an explicit "not a certified mechanic" notice and linking to trusted resources.

7.2 Four Principles

Context Area	Beneficence	Nonmaleficence	Autonomy	Justice
Public Health	Alerts users to urgent car issues	Avoids harm by giving timely diagnostics	Users choose whether to act on alerts	Applies same diagnostics to all users
Global/Social	Empowers non-technical users	Avoids misdiagnosis through fallback logic	Users get understandable car data	Accessible UI regardless of skill/background
Environmental	Reduces emissions via proactive fixes	Avoids waste by detecting early problems	Users decide how to maintain their car	Promotes sustainability affordably
Economic	Saves users money on repairs	Avoids unnecessary mechanic visits	Users control costs through information	Free tier ensures access for all

Important Area: **Economic - Beneficence**. We specifically designed FixIT to be affordable, even offering a free tier. This helps financially constrained users perform diagnostics they'd otherwise skip.

Lacking Area: **Environmental - Nonmaleficence**. Our current system doesn't directly offset environmental harm. In the future, we could integrate eco-driving feedback and emissions reporting to improve our environmental impact.

7.3 Virtues

Team-Wide Virtues:

1. **Transparency** – We openly discuss design decisions, challenges, and capabilities with our users and teammates.

2. **Perseverance** – We tackled complex subsystems like BLE and cloud integration persistently despite setbacks.
3. **Collaboration** – We assigned tasks based on strengths, supported each other during testing phases, and adjusted workloads fairly.

Individual Reflections:

Mohamed Elaagip

- Demonstrated Virtue: **Diligence**
    - Important because building reliable user-facing software requires attention to detail.
    - Demonstrated through repeated frontend revisions and responsive updates based on tester feedback.
- Undemonstrated Virtue: **Patience**
    - Important in debugging and testing complex features without frustration.
    - Plan to improve by setting time-buffered debugging goals and documenting frustrations for better future mitigation.

Benjamin Muslic

- Demonstrated Virtue: **Curiosity**
    - Important because exploring multiple OBD protocols helped solve low-level hardware issues.
    - Demonstrated by implementing and debugging ISO 9141-2 and CAN protocol support.
- Undemonstrated Virtue: **Humility**
    - Important for seeking feedback earlier.
    - Could improve by reviewing pull requests with others before merging.

Jonathan Duron

- Demonstrated Virtue: **Adaptability**
    - Important when building React Native components that must support varied mobile devices.
    - Demonstrated by refactoring BLE components multiple times to address user issues.
- Undemonstrated Virtue: **Confidence**
    - Important to advocate for design ideas.
    - Can improve by proposing and defending UI/UX refinements proactively.

William Griner

- Demonstrated Virtue: **Responsibility**
    - Critical for managing AWS resources securely and reliably.
    - Demonstrated by owning infrastructure-as-code setup and maintaining Terraform locking.

- Undemonstrated Virtue: **Creativity**
    - Important for optimizing backend workflows.
    - Could improve by brainstorming unique API endpoint formats or analytics features.

Overall, our team upheld professional responsibility through deliberate testing, transparent communication, and a commitment to privacy and usability. There's room to improve on proactive environmental responsibility and design advocacy, which we plan to address in future iterations.

8 Conclusions

8.1 Summary of Progress

FixIT successfully combines embedded automotive diagnostics, cloud-based AI services, and mobile interfaces into a single system that empowers users to understand and manage their car's maintenance needs. Over the course of this project, we built and tested a working pipeline from an ESP32-based OBD-II reader to a React Native app, supported by AWS Lambda backend services and OpenAI GPT integration.

We overcame challenges related to BLE instability, AI response control, and UI design, and gained valuable experience in cross-disciplinary collaboration, system architecture, and ethical software development. Each team member grew in their respective technical areas while contributing to a common, impactful goal.

Moving forward, we see opportunities to extend FixIT's capabilities—such as adding emissions tracking, advanced alert analytics, and integrations with mechanic scheduling services. Ultimately, we believe FixIT represents a meaningful step toward more accessible and intelligent car care, especially for users without deep technical knowledge or resources.

8.2 Value Provided

Our design provides clear value by addressing real needs faced by everyday vehicle owners. Many drivers struggle to interpret Diagnostic Trouble Codes (DTCs), leading to unnecessary trips to mechanics or delays in critical repairs. FixIT directly addresses this by offering users timely and accessible information about their vehicle's condition through a user-friendly mobile app.

Addressing the Original Problem

FixIT empowers users with real-time insights into their car's health. By reading DTCs via an ESP32 OBD-II interface and displaying them in readable form through our mobile app, users gain clarity about what's happening under the hood. This addresses the core need for transparency and control in vehicle maintenance, especially for non-technical drivers.

Addressing the Original Problem

We set out to make vehicle diagnostics easier to understand and more accessible, and we have delivered on that core objective. Our system successfully extracts and transmits data from a real vehicle using our custom hardware and Bluetooth communication pipeline. Although our AI-based code interpretation is not yet fully integrated, the current version already processes raw data and informs users of active trouble codes—an essential first step in the solution.

Fit in the Broader Context

In the broader context of automotive maintenance, FixIT contributes to the growing trend of preventative diagnostics, smart vehicle monitoring, and user empowerment. It aligns with current movements toward connected cars and consumer-first vehicle tech. By bridging the gap between vehicle-generated data and user understanding, FixIT promotes informed decision-making, reduced repair costs, and potentially better long-term vehicle performance.

As we continue to develop AI features that explain DTCs using real-world data and community-sourced interpretations, the value of FixIT will only grow, positioning it within a broader shift toward intelligent, accessible, and proactive car maintenance tools.

8.3 Next Steps

Short-Term Improvements

1. Hardware Integration

- Design custom PCB to reduce form factor and power consumption
- Add status LEDs for easier troubleshooting

2. User Authentication and Payment

- Implement secure user sign-up and login flow
- Add in-app payments for premium features
- Develop subscription model for ongoing service

3. Cloud Infrastructure Enhancements

- Implement multi-region failover (beyond current multi-availability architecture)
- Optimize Lambda cold starts for faster response times
- Add comprehensive monitoring and alerting

Medium-Term Roadmap

1. Enhanced Diagnostic Capabilities

- Support for ABS, airbag, and transmission-specific DTCs
- Integrate with parts pricing APIs for repair cost estimates
- Add vehicle-specific service interval recommendations

- 2. Data Analytics
  Build predictive maintenance models using machine learning
- Provide fleet-level insights for users with multiple vehicles
- Develop trend analysis for recurring issues

3. Integration Ecosystem

- Create API for third-party service providers
- Develop mechanic booking integration
- Build vehicle history reporting for resale value estimation

Business Development

1. Marketing Strategy

- Target DIY car enthusiasts through automotive forums and social media
- Partner with auto parts retailers for co-promotion
- Develop educational content on vehicle maintenance

2. Customer Support

- Create comprehensive knowledge base for common issues
- Implement in-app support chat
- Develop community forum for user discussions

3. Revenue Expansion

- Explore B2B opportunities with fleet management companies

- Consider white-label solutions for automotive brands
- Develop referral partnerships with repair shops

9 References

1. C. Bhat and F. S. Koppelman, "A conceptual framework of individual activity program generation," Transportation Research Part A: Policy and Practice, vol. 27, no. 6, pp. 433-446, 1993.

2. ISO, "Road vehicles — Communication between vehicle and external equipment for emissions-related diagnostics — Part 4: External test equipment," ISO 15031-4:2014, 2014.

3. J. P. Stokes, "Bluetooth low energy security: a quantitative analysis of pairing, data signing, and passive eavesdropping," Ph.D. dissertation, Dept. Comp. Sci., Univ. of Oxford, Oxford, UK, 2019.

4. M. Kumar and Y. H. Kim, "The automotive industry: Economic impact and location issues," Regional Economics: Theory and Practice, vol. 36, no. 5, pp. 521-533, 2020.

5. S. E. Mathews, "React Native: Building mobile apps with JavaScript," O'Reilly Media, Inc., 2022.

6. N. Suryadevara and S. C. Mukhopadhyay, "Internet of Things: A review and future perspective," Wireless Networks, vol. 23, pp. 1-11, 2017.

7. T. Brown, et al., "Language models are few-shot learners," Advances in Neural Information Processing Systems, vol. 33, pp. 1877-1901, 2020.

8. AWS, "AWS Lambda Developer Guide," Amazon Web Services, Inc., 2023. [Online]. Available: https://docs.aws.amazon.com/lambda/

9. Espressif Systems, "ESP32 Technical Reference Manual," Version 4.4, 2021.

10. OBD Solutions, "ECUsim 2000 User Manual," Version 2.5, 2021.

10 Appendices

Any additional information that would be helpful to the evaluation of your design document.

If you have any large graphs, tables, or similar data that does not directly pertain to the problem but helps support it, include it here. This would also be a good area to include hardware/software manuals used. May include CAD files, circuit schematics, layout etc,. PCB testing issues etc., Software bugs etc.

Appendix 1 – Operation Manual

Table of Contents

System Requirements

Development Environment

- macOS/Windows/Linux operating system

- Node.js (v18 or higher)
- npm (v8 or higher)
- Git for version control
- Xcode (for iOS development, macOS only)
- Android Studio (for Android development)
- Arduino IDE (for ESP32 programming)
- AWS Account (for backend infrastructure)

Mobile Requirements

- iOS: iPhone running iOS 13.0 or later
- Android: Phone running Android 8.0 (API level 26) or later
- Bluetooth: Bluetooth 4.2 or later with BLE support

Hardware Requirements

- ESP32 microcontroller
- OBD-II Scanner with ELM327 compatibility
- OBD-II cable compatible with your vehicle

Repository Setup

1. Clone the FixIT repository from GitHub: git clone https://github.com/willyg23/predictive_car_maintenance_SD.git cd predictive_car_maintenance_SD/frontend/FixIt
2. Install the root dependencies: npm install

Environment Configuration

Frontend Environment Setup

1. Create a .env file in the frontend/FixIT directory with the following variables: OPENAI_API_KEY=your_openai_api_key BACKEND_API_URL=your_backend_api_url
2. For iOS development, set up CocoaPods: cd frontend/FixIT/ios pod install cd ../..

Backend Environment Setup

Backend services are deployed on AWS. If you need to deploy your own instance:

1. Install the AWS CLI and authenticate with your AWS account
2. Navigate to the terraform directory
3. Initialize Terraform: terraform init
4. Apply the configuration: terraform apply

Application Development Setup

Frontend Setup

1. Navigate to the frontend FixIT directory: cd frontend/FixIT
2. Install the required dependencies: npm install

ESP32 Configuration

1. Open the Arduino IDE
2. Install ESP32 board support by adding this URL to your Additional Board Manager URLs: https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json
3. Go to Tools > Board > Boards Manager, search for ESP32 and install the latest version
4. Install the required libraries:
    a. BLE
    b. WiFi
    c. ArduinoJson
5. Open the ESP32 sketch from embedded/ESP32-Code/ESP32-Code.ino
6. Connect your ESP32 to your computer via USB
7. Select the correct board and port
8. Upload the sketch to your ESP32

Running the Application

iOS Simulator

1. Navigate to the FixIT frontend directory: cd frontend/FixIT
2. Start the Metro bundler: npm start
3. In a new terminal, run the iOS app in the simulator: npm run ios

Android Emulator

1. Start an Android emulator from Android Studio (must have Bluetooth capability)
2. Navigate to the FixIT frontend directory: cd frontend/FixIT
3. Start the Metro bundler: npm start
4. In a new terminal, run the Android app: npm run android

Physical Devices

iOS Device

1. Open the Xcode project in the ios folder
2. Select your device as the build target
3. Sign the app with your Apple Developer account

4. Build and run the app

Android Device

1. Enable USB debugging on your Android device
2. Connect the device to your computer
3. Run npm run android to install and launch the app

HARDWARE CONNECTION

Connecting the ESP32 to the OBD-II Port

1. Ensure the ESP32 has been programmed with the FixIT firmware
2. Connect the ESP32 to the ELM327 OBD-II scanner
3. Locate the OBD-II port in your vehicle (typically under the dashboard)
4. Connect the OBD-II scanner to your vehicle's OBD-II port
5. Power on your vehicle to at least the accessory position to power the ESP32

Pairing with the Mobile App

1. Launch the FixIT app on your mobile device
2. Navigate to the Bluetooth Scanner screen
3. The app will automatically scan for FixIT-compatible devices
4. Select your ESP32 device from the list of available devices
5. Once connected, the app will show a confirmation message
6. The app will automatically begin receiving vehicle data

USING THE FIXIT APPLICATION

Initial Setup

1. After installing the app, you'll be prompted to add a vehicle
2. Enter the required information:
    a. Year, Make, and Model
    b. Current mileage
    c. Last oil change date
    d. Vehicle nickname (optional)
3. Tap "Save" to create your vehicle profile

Vehicle Management

1. Access the Car Carousel to view all your vehicles
2. Swipe left or right to switch between vehicles
3. Tap the "+" button to add a new vehicle
4. Long-press a vehicle card to edit or delete a vehicle

Diagnostic Functions

1. Connect your OBD-II scanner as described above
2. Navigate to the Bluetooth Scanner screen
3. Once connected, the app will automatically:
    a. Read DTCs (Diagnostic Trouble Codes)
    b. Monitor engine temperature
    c. Check for maintenance issues
4. Detected issues will be displayed with:
    a. A human-readable explanation
    b. Urgency level (color-coded)
    c. Recommended actions

Maintenance Tracking

1. View your vehicle's maintenance status on the dashboard
2. Color indicators show status:
    a. Green: Good condition
    b. Yellow: Maintenance due soon
    c. Red: Maintenance overdue
3. Tap on any status indicator for detailed information
4. Add service records by tapping "Add Service" on the Service History screen

AI Chat Assistant

1. Navigate to the Chat tab
2. Type your vehicle-related question
3. The AI will respond with information specific to your vehicle
4. The chat supports questions about:
    a. Maintenance intervals
    b. Explaining warning lights
    c. Cost estimates for repairs
    d. General vehicle information

TROUBLESHOOTING

Bluetooth Connection Issues

- Ensure Bluetooth is enabled on your mobile device
- Verify the ESP32 is powered and running the correct firmware
- Reset the ESP32 by unplugging it and plugging it back in
- Ensure your mobile device supports Bluetooth Low Energy (BLE)
- On Android, verify location permissions are enabled (required for BLE scanning)

App Crashes

- Ensure you have the latest version of the app
- Clear the app cache:
    - iOS: Settings > General > iPhone Storage > FixIT > Offload App
    - Android: Settings > Apps > FixIT > Storage > Clear Cache
- Reinstall the application

OBD-II Connection Problems

- Verify your vehicle is OBD-II compliant (most vehicles after 1996)
- Ensure the vehicle is powered on to at least the accessory position
- Try a different OBD-II port if your vehicle has multiple
- Test the OBD-II scanner with another device to verify functionality

Backend Service Unavailability

- Check your internet connection
- The app will continue to function in offline mode with limited features
- Core diagnostic features should work without internet connectivity

Maintenance and Updates

Firmware Updates

If new ESP32 firmware becomes available:

1. Download the latest firmware from the repository
2. Connect the ESP32 to your computer via USB
3. Open the Arduino IDE with the new firmware
4. Upload the firmware to your ESP32

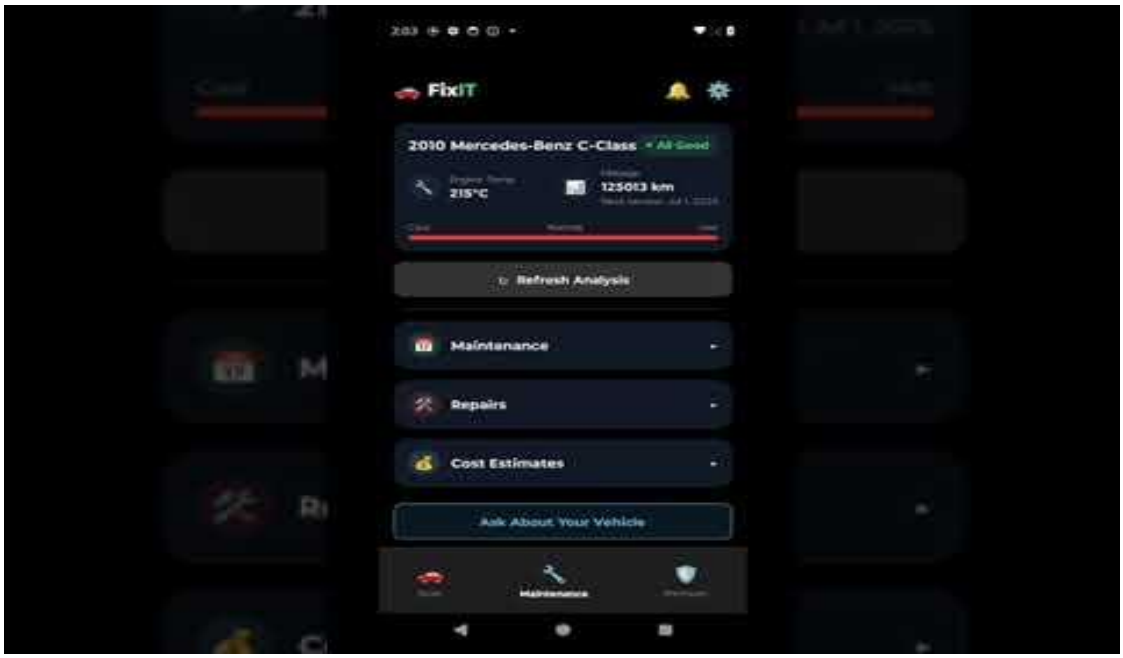
Cloud Infrastructure Updates

If you're running your own backend infrastructure:

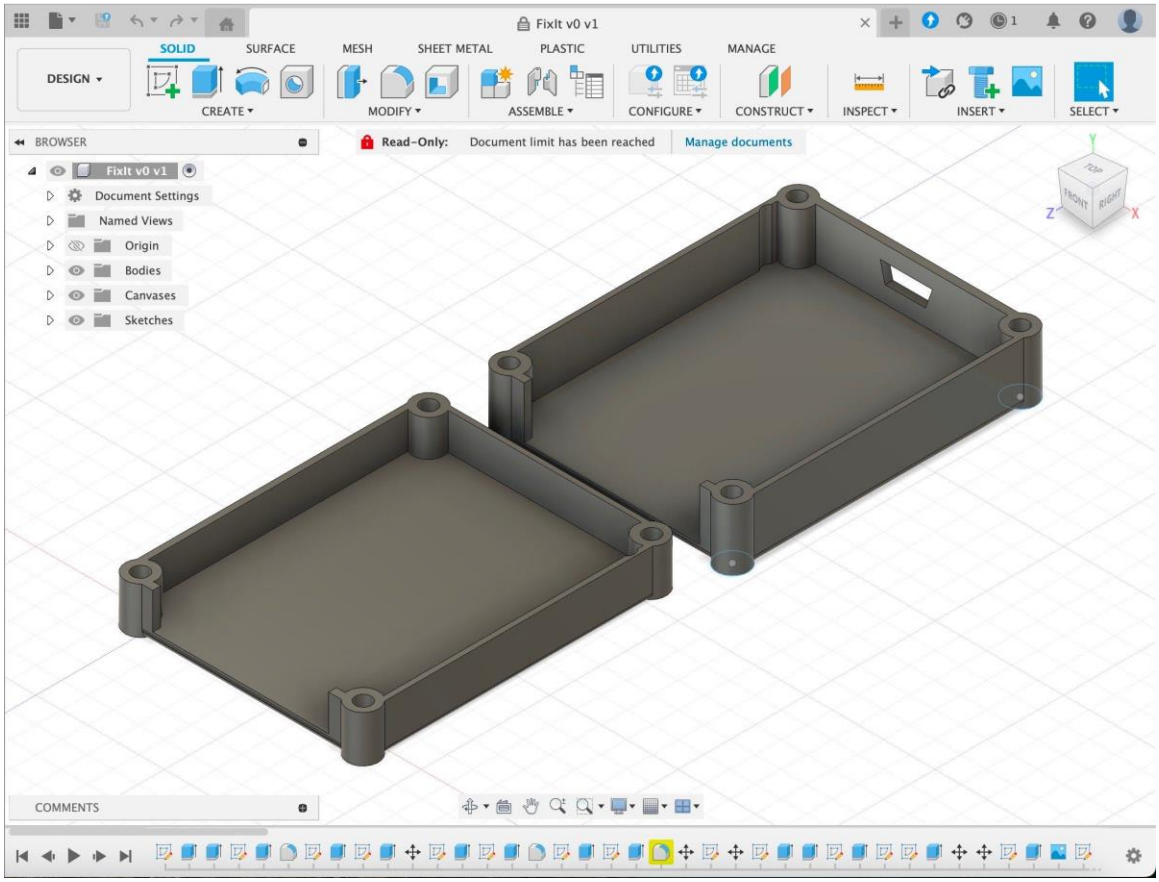
1. Pull the latest changes from the repository
2. Navigate to the terraform directory
3. Run terraform apply to update the infrastructure

App workflow:

https://youtu.be/RymSTPquqBA?si=BR9HaboLMph5pfvO



CAD file image:



APPENDIX 2 – ALTERNATIVE/INITIAL VERSION OF DESIGN

1. Versions considered before client's specifications have changed

Since we are our own client, not much is relevant for this section.

2. Versions considered before learning more about the project

We initially considered making our own ML model that can predict car maintenance issues, but then realized we don't have access to an adequate data set to make that a reality.

The Original Plan for our ML model

- Given a car's: make, model, year, mileage, and an error code that is occurring, the ML model can accurately predict what part or parts need to be replaced to resolve the error code.
- For example: a 2005 chevy malibu with 50k miles has an error code that says cylinder misfire. the cylinder misfire could be caused by this part or parts being faulty: spark plug, fuel injector, ignition coil, and more. well, that error message only says cylinder misfire, not what part or parts need to be replaced.
- The ML model would solve this problem by telling the user which part is most likely to be causing the cylinder misfire. This is possible as some cars are know to consistently have the same problems, i.e. the 2002-2006 Honda CRVs with their air conditioning compressors
- The ML model would be a classification model, multilabel or multiclass, depending on how we structure the data.

The data we'd need to train said model, at the minimum:

- Make
- Model
- Year
- Mileage
- What error code(s) have occurred
- What mileage the error code(s) occurred at
- Which specific part(s) needed to be fixed to resolve the issue presented by the corresponding error code(s). This is our target variable.

We do not have access to a dataset with this information.

It's worth noting that for this to work consistently for even one make and model of one car, we'd need hundreds if not thousands of examples of data. Let's say 1000 examples per car (Which could be an underestimate). Now if we want to make an app that works for many cars? Well, there's millions of variations of cars in the world. Let's say there's a million. 1 million * 1000 == 1 billion, and we simply don't have access to a dataset of this size.

Since we don't have a preexisting dataset, we'd have to get all this incredibly specific and correctly labeled data ourselves, which is not feasible for our team. Not to mention, there could be more factors in the dataset that we'd need to produce an accurate model. Such as: driver behavior, region(s) where the car has been driven and how long it's been driven in each region, manufacturing recalls, and much more.

3. Versions that resulted in failure to achieve specifications, etc.

We did not have any major versions that failed to achieve specifications. Of course we had bugs to fix along the way, but we didn't have a major version release that failed.

4. Describe each major design version and why they were scrapped/revised

The only major design version that was scrapped was using EC2 for compute, and EC2 was swapped for lambda. As mentioned in the previous section 6:

Originally, we started with a more traditional cloud backend using EC2 instances, auto-scaling groups, and load balancers. This setup worked — we were able to deploy code, scale our servers, and serve backend traffic — but it came with a lot of complexity. We had to manage AMIs, configure ALBs (Application Load Balancers), write CI/CD pipelines for instance provisioning, and handle SSH access for debugging.

Eventually, we decided to pivot to a serverless model using AWS Lambda. The switch simplified our deployment process dramatically. Lambda gave us built-in auto-scaling, seamless integration with API Gateway, and a pay-as-you-go model that fit our needs better. There was no longer a need to configure a load balancer or manually manage compute resources, as Lambda scales automatically. Deployment became as simple as pushing a new Docker image to AWS.

We considered this shift a major win. It made the backend more maintainable, easier to understand, and reduced the amount of DevOps work we had to do. The EC2-based system became a "completed but scrapped" sub-system — we fully implemented it, got it working, and then replaced it because Lambda offered a better path forward.

Appendix 3 – Other considerations

- Any miscellany you deem important, what you learned, anything funny, anecdotes from your project experience

Not much to put here. We had fun building our project. and we learned a lot about the areas we each worked on along the way.

Appendix 4 – Code

- Include your GitHub repository link, snippets of code, etc.

Github repo link: https://github.com/willyg23/predictive_car_maintenance_SD

Appendix 5 – Team Contract

Complete each section as completely and concisely as possible. We strongly recommend using tables or bulleted lists when applicable.

Team Contract

Team Members:

1) _____Benjamin Muslic_____ 2) _____Jonathan Duron _____

3) _____William Griner_____ 4) _____Mohammed Elaagip _____

Team Procedures

1. Day, time, and location (face-to-face or virtual) for regular team meetings:

   Since the start of the semester our meeting time has been after Thursday's class. It makes the most sense to us especially if class ends early and our mind is focused on senior design instead of other things. We consider this our official weekly in-person tag up. Tuesday 2:30 are meetings with our advisor.

2. Preferred method of communication updates, reminders, issues, and scheduling (e.g., e-mail, phone, app, face-to-face):|

   Discord and/or other apps

3. Decision-making policy (e.g., consensus, majority vote):

   Discussions and debates until we come to a mutual consensus

4. Procedures for record keeping (i.e., who will keep meeting minutes, how will minutes be shared/archived):

   Reserving a conference room and writing on whiteboard. Take pictures at the end and publishing them on notion that is shared with the team.

Participation Expectations

1. Expected individual attendance, punctuality, and participation at all team meetings:

   All team members are expected to attend every meeting, arrive on time, and

actively contribute to discussions.

2. Expected level of responsibility for fulfilling team assignments, timelines, and deadlines:

   Each member is responsible for completing their assigned tasks on time and ensuring that the team meets its overall deadlines.

3. Expected level of communication with other team members:

   Members should maintain open, consistent communication and promptly update the team on progress, issues, or any delays.

4. Expected level of commitment to team decisions and tasks:

   Team members are expected to support and contribute fully to team decisions and actively work towards completing all tasks

Leadership

1. Leadership roles for each team member (e.g., team organization, client interaction, individual component design, testing, etc.):
2. Strategies for supporting and guiding the work of all team members:

Regular Check-ins and Progress Updates: Schedule consistent team meetings or check-ins to review each member's progress, address any challenges, and offer guidance or resources when needed.

Collaborative Problem-Solving: Foster an environment where team members feel comfortable seeking help and where the group collaborates to solve issues, ensuring everyone has the support they need to meet deadlines and deliver high-quality work.

Quality Documentation: Write down our thoughts in an easy-to-understand way, so our whole team can understand the technicalities of what we're doing.

3. Strategies for recognizing the contributions of all team members:

   Public Acknowledgment: Regularly acknowledge individual contributions during team meetings or through team-wide communications to highlight efforts and celebrate successes.

Rotating Spotlight or Recognition Programs: Implement a system where team members can nominate each other for contributions, or rotate a "spotlight" that focuses on recognizing one team member's efforts each week. Similar to how morning standups are done in professional environment.

Project Management Style Adopted by the team

We chose a mix of waterfall and agile. We develop most of our features in the waterfall style, while occasionally stopping to see what we could improve on, and what me might want to change.

Collaboration and Inclusion

1. Describe the skills, expertise, and unique perspectives each team member brings to the team.

Benjamin Muslic:

- Embedded
- Hardware
- Front-end app development
- Automotive knowledge
- C
- 3D Printing

Jonathan Duron:

- React Native
- Frontend app development
- UI Design
- Cloud Engineering

Mohamed Elaagip:

- Embedded
- UI Design
- Hardware
- Front-end development
- Signals
- FPGA

William Griner:

- Basic skills in AI / ML, Data science, and Data labeling
- Cloud Engineering
- Python
- Flask

2. Strategies for encouraging and supporting contributions and ideas from all team members:

    - Jon's experience in developing apps will help us stay on track and make sure we meet our deadlines.
    - Will's Python, Flask, and Cloud computing knowledge will come in handy in setting up our backend and organizing and aggregating our data. Will's ML and data science knowledge will be helpful for deciding whether or not we want to create our own ML model, or leverage existing models.
    - Mohamed's low-level skills will be useful in our OBD endeavors, and his UI skills will be useful when designing our UI.
    - Ben's hardware experience and embedded knowledge will help us understand the computer system of the vehicle and obtaining the necessary data.

3. Procedures for identifying and resolving collaboration or inclusion issues (e.g., how will a team member inform the team that the team environment is obstructing their opportunity or ability to contribute?)

Goal-Setting, Planning, and Execution

1. Team goals for this semester:
   Have a functional prototype by the end of the first semester to show our advisor.
   Keep open communication
2. Strategies for planning and assigning individual and team work:

   By the end of the meeting we will discuss what our plans are for the week and assign work from there.

3. Strategies for keeping on task: meetings and internal demos

Having check ins 2 days out of the week to keep track of progress.

Consequences for Not Adhering to Team Contract

1. How will you handle infractions of any of the obligations of this team contract?

   We will discuss with the team members the whole team about the problem.

   We will address issues to our advisor

2. What will your team do if the infractions continue?

   We will have to report to the professor and ask how we should proceed the problem.

****

a) *I participated in formulating the standards, roles, and procedures as stated in this contract.*

b) *I understand that I am obligated to abide by these terms and conditions.*

c) *I understand that if I do not abide by these terms and conditions, I will suffer the consequences as stated in this contract.*

1) _____Benjamin Muslic_____ DATE _____9/19/2024_____

2) _____Jonathan Duron_____ DATE _____9/19/2024_____

3) _____Mohamed Elaagip_____ DATE _____09/19/2024_____

5) _____ William Griner_____ DATE _____09/19/2024_____