REPULBLIC OF CAMEROON

PEACE-WORK-FATHERLAND

UNIVERSITY OF BUEA

FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER ENGINEERING

SRS Document

Design and Implementation of a Mobile Application for Car Fault Diagnosis

Course Information

Course Title: Internet and Mobile Programming

Course Code: CEF440

Course Instructor: Dr. Nkemeni Valery

Group Members

LEKEUGO DEMELIEU ROCHINEL	FE22A237
CHUYE PRINCELY TATA	FE22A184
DIONE CHANCELINE NZUO SONE	FE22A187
ACHUO ESEGNI	FE22A135
MISHAEL ZABUD	FE22A248

Table of Contents

Software Requirements Specification (SRS)

Project Title: Design and Implementation of a Mobile Application for Car Fault Diagnosis

Prepared by: Group 19

I.Introduction

➢ Purpose

The purpose of this document is to outline the complete software requirements for the Car Fault Diagnosis Mobile Application. This application aims to assist car owners in diagnosing faults using dashboard light recognition and engine sound analysis.

➢ Scope

The mobile application will use smartphone sensors (camera and microphone) and artificial intelligence models to:

- ✓ Identify dashboard light symbols
- ✓ Analyze engine sounds
- ✓ Suggest probable faults and solutions
- ✓ Provide tutorial links and recommendations
- ✓ Work offline for basic functionality

➢ Definitions, Acronyms, and Abbreviations

- ✓ **AI**: Artificial Intelligence
- ✓ **ML**: Machine Learning
- ✓ **API**: Application Programming Interface
- ✓ **SRS**: Software Requirements Specification
- ✓ **UI/UX**: User Interface/User Experience
- ✓ **React Native**: Framework for building cross-platform mobile apps

➢ User Characteristics

- ✓ Car owners: Minimal technical knowledge
- ✓ Mechanics: Moderate technical knowledge
- ✓ Developers: Experienced in React Native and Python

➢ Operating Environment

- ✓ Android and iOS mobile devices
- ✓ Internet connection for advanced features
- ✓ Offline mode for limited usage

➢ Assumptions and Dependencies

- ✓ Users will have modern phones
- ✓ Backend services will be continuously available

➢ References

- ✓ IEEE SRS Standard Template
- ✓ Stakeholder Interviews and Questionnaires
- ✓ System Modeling Documentation

➢ Overview of the Document

This document includes requirement review and analysis, classification, prioritization, SRS details, and validation strategy.

II. System Architecture and Technology Stack

Overview of System Architecture

The system follows a **modular and service-oriented architecture** that separates the mobile frontend from the AI-powered backend. This design allows the system to be scalable, maintainable, and easy to extend.

The architecture comprises the following key layers:

1. **Mobile Frontend (React Native)**:
   Cross-platform mobile application providing user interface and handling camera/audio input from the user.
2. **Backend Server (API Layer)**:
   Responsible for routing requests from the mobile app to appropriate AI services or databases. This also manages user accounts, diagnostic records, and third-party API integrations.
3. **AI Microservices (Python Models)**:
   Deployed as standalone REST APIs, these services handle image-based and audio-based fault diagnosis using trained models (e.g., TensorFlow, PyTorch).
4. **Cloud Storage & Database**:
   Stores diagnostic records, logs, user data, and model results.
5. **Optional Third-Party Integrations**:

   - ✓ YouTube Data API for educational video content
   - ✓ Notification services (e.g., Firebase Cloud Messaging)

Technology Stack Breakdown

Frontend (Mobile Application)

Component	Technology
Framework	**React Native** (JavaScript/TypeScript)
Navigation	React Navigation
Camera Access	React-native-camera
Audio Recording	React-Rative-Audio
HTTP Requests	Fetch API
Platform Compatibility	Android and iOS

Backend (API & Logic Layer)

Component	Technology
Runtime Environment	**Python Flask / FastAPI**
REST API Development	Flask / FastAPI for Python-based model services
Authentication	JWT (JSON Web Tokens) or Firebase Auth
Hosting	Render, Heroku, or custom VPS (e.g., DigitalOcean)

AI & Machine Learning Models

Component	Technology
Image Recognition	TensorFlow, PyTorch, or OpenCV
Audio Analysis	LibROSA, TensorFlow Audio, or custom CNN/RNN models
Model Deployment	Flask/FastAPI serving as RESTful endpoints
Training Tools	Jupyter Notebook, Google Colab, Python scripts
Data Handling	Pandas, NumPy, Scikit-learn

Storage & Database

Component	Technology
User & App Data	PostgreSQL
File Storage	Firebase Storage
ORM	SQLAlchemy (Python)

ORM stands for **Object-Relational Mapping**. It is a programming technique used to interact with a **relational database** (like MySQL, PostgreSQL, etc.) using **objects** in your

programming language (such as Python, JavaScript, etc.), instead of writing raw SQL queries.

Notifications & Messaging

Component	Technology
Push Notifications	Firebase Cloud Messaging (FCM)

III. Requirement Analysis

Requirement Analysis is the process of refining and validating the gathered requirements to ensure they are complete, clear, feasible, and well-structured. This phase bridges the gap between requirement gathering and system design.

Key Objectives:

- Review and analyze the gathered requirements for completeness, clarity, and feasibility.
- Identify inconsistencies, ambiguities, and missing information.
- Prioritize requirements based on importance and feasibility.
- Classify requirements (functional vs. non-functional).
- Develop a Software Requirement Specification (SRS) document.
- Validate requirements with stakeholders.

1. Review and Analyze Gathered Requirements

A. Completeness Check

Definition:

Completeness refers to whether **all necessary requirements** have been gathered and documented for the system to function correctly.

A completeness check ensures that every functional and non-functional requirement needed to fully describe the system's behavior is present. This includes all user scenarios, system actions, validations, and business rules. Missing features or overlooked edge cases would indicate an incomplete requirement set

Below are some completeness checks gotten from our requirements gathering process

- ## Dashboard Light Recognition:

Problem: 68.8% of users consult mechanics because they don't understand symbols.

Missing Details:

- ✓ Which dashboard symbols? (Check Engine, ABS, Oil Pressure, etc.)
- ✓ How to handle variations? (Different car manufacturers use slightly different icons.)
- ✓ Real-time vs. photo upload? Should the app scan live or analyze a saved image?

- ## Engine Sound Analysis:

Problem: 80% of users can't diagnose abnormal sounds.

Missing Details:

- ✓ What sounds to detect? (Knocking, squealing, hissing, grinding.)
- ✓ Background noise handling? (Will the app filter out wind/traffic noise?)
- ✓ Recording duration? (Should users record 5 sec, 10 sec, or continuous monitoring?)

- ## AI Assistance (100% Demand)

Problem: Users want AI but don't know how it works.

Missing Details:

- ✓ Explainable AI? Should the app say, "This sound matches a worn-out belt (85% confidence)"?
- ✓ False positives? What if the app wrongly diagnoses a serious issue?

- ## Offline Functionality:

Problem: Users in rural areas may lack internet.

Missing Details:

- ✓ What works offline? Only basic symbol recognition, or also cached sound analysis?
- ✓ Storage limits? How many preloaded models can the app store?

- **Video Tutorials & Mechanic Linking**

Problem: Users want expert validation.

Missing Details:

- ✓ YouTube API vs. Embedded Videos? Should videos play in-app or redirect?
- ✓ Mechanic verification? How to ensure recommended mechanics are trustworthy?

B. Clarity Check

Definition:
Clarity means that each requirement is **understandable, unambiguous, and well-defined**, leaving no room for misinterpretation.

A clarity check ensures that requirements are written in clear, simple language and avoid vague terms like "fast", "better", or "user-friendly" without proper metrics or definitions. Well-clarified requirements help both developers and stakeholders have a common understanding of what is to be built

We also did some clarity checks based on our requirement gathering as below

- **Provide YouTube Video Suggestions**

Ambiguity: Should the app:

- ✓ Search YouTube dynamically ("How to fix knocking sound")?
- ✓ Use a pre-approved playlist (curated by mechanics)?
- ✓ Embed videos directly (requires more storage)?

- **Confidence Scores for Predictions**

Ambiguity:

- ✓ Numerical (85%) vs. Qualitative (High/Medium/Low)?
- ✓ Should low-confidence results still show recommendations?

- **Offline Functionality**

Ambiguity:

- ✓ Cache recent scans? (Store last few diagnoses offline.)
- ✓ Preload common sounds/symbols? (But increases app size.)

C. Technical Feasibility

Definition: Technical feasibility assesses whether the requirements can be implemented within the project's constraints (time, budget, technology).

Evaluation: using

- **Feasible Requirements:**
  - ✓ AI models for image/sound recognition (TensorFlow, Python).
  - ✓ Offline mode (caching data locally).
- **Risky/Challenging Requirements:**
  - ✓ Real-time mechanic chat (requires robust backend infrastructure).
  - ✓ Bluetooth OBD-II integration (depends on external hardware).

Resolutions:

The following resolutions were then made:

- Prioritize phone-based features (e.g., camera/microphone inputs) over hardware-dependent ones.

Phase out complex features (e.g., mechanic chat) for future updates.

Requirement	Feasibility Analysis	Risks & Mitigations
Dashboard Light Recognition	Uses **OpenCV + TensorFlow Lite** (mobile-friendly).	**Risk:** Low light conditions reduce accuracy. **Fix:** Add "Use flash" prompt.
Engine Sound Classification	**CNN + MFCCs** work for audio ML.	**Risk:** Noisy environments distort recordings. **Fix:** Add noise suppression.
Offline Mode	**Pre-trained TensorFlow Lite models** can run offline.	**Risk:** Large model size. **Fix:** Quantize models to reduce size.

YouTube API Integration	YouTube Data API allows search & embedding.	**Risk:** API changes may break features. **Fix:** Monitor updates.
Mechanic Connection	Needs **backend API + payment system**.	**Risk:** Legal/trust issues. **Fix:** Partner with verified garages.

D. Dependency Relationships

Definition: Dependency relationships identify which requirements must be implemented before others.

Evaluation:

- **Core Dependencies:**
  - AI model training must precede app integration.
  - Offline mode requires local storage setup before implementation.
- **Secondary Dependencies:**
  - ✓ YouTube API integration depends on a stable internet connection.
  - ✓ Mechanic contact feature requires user authentication.

Resolutions:

We then decided to:

- ✓ Develop a phased implementation plan (e.g., AI models first, UI second, advanced features last).
- ✓ Use a Gantt chart to visualize task dependencies.

Dependency	Explanation	Example
Sound & Light Recognition → Recommendation System	AI models must first detect a fault before suggesting repairs.	Knocking sound detected → Recommend "Check engine oil or timing belt."
Offline Mode → Pre-trained Model Optimization	Offline use requires models to be small and efficient.	Quantized TensorFlow Lite models for symbol recognition.

Mechanic Connection → User Auth & Payments	Secure transactions and verified access require login and payment support.	Firebase Auth for identity + Stripe API for payment.

2. Inconsistencies, Ambiguities, and Missing Information

A. Inconsistencies:

Definition:
Inconsistencies occur when **two or more requirements contradict each other** or imply different behaviors.

For example, if one requirement says the system must operate offline, but another says it must always connect to a cloud server, that's inconsistent. These conflicts must be identified and resolved to avoid confusion during development.

From our Requirements, we have:

- **Survey vs. Interviews**
- ✓ **Conflict:** Some users want standalone AI, others want mechanic integration.
- ✓ **Solution:** Make mechanic linking optional (not forced).
- **Reverse Engineering Findings**
- ✓ **Conflict:** Most apps use OBD-II dongles, but we use AI based mobile phones.
- ✓ **Solution:** Emphasize hardware-free advantage in marketing.

B. Ambiguities:

Definition:
Ambiguities are unclear or vague statements in the requirements that can be interpreted in multiple ways.

For example, a requirement that states "The system shall provide quick access to fault details" is ambiguous because "quick" is subjective. Should it load in 1 second? 5 seconds? Ambiguities often lead to misaligned expectations and flawed implementations.

We also found some Ambiguous Requirements in work, and we delt with them as shown:

- **Basic Offline Functionality**

Clarified Requirement:

- ✓ The app shall recognize a few most common dashboard symbols offline.
- ✓ The app shall allow recording engine sounds offline but require internet for analysis.

- **Automated Explanations**

Clarified Requirement:

- ✓ The app shall provide text + visual icons for explanations.
- ✓ Critical warnings shall include voice alerts (e.g., 'Stop engine immediately').

C. Missing Information:

Definition:
This refers to **gaps in the requirements** where critical data or logic has not been specified.

Missing information could involve unspecified inputs, undefined data formats, lack of error-handling scenarios, or unclear user roles. These gaps must be filled through follow-up discussions with stakeholders or domain experts to ensure the system behaves correctly under all conditions.

Here, we have:

- **Error Handling**

New Requirement:

- ✓ If the AI confidence is below 70%, the app shall display: 'Unable to diagnose. Try again in a quieter environment or consult a mechanic.

- **User Feedback Mechanism**

New Requirement:

- ✓ Users shall rate diagnoses and submit corrections.

- **Data Privacy**

New Requirement:

- ✓ Engine sound recordings shall be deleted after 24 hrs unless saved by the user.

3. Prioritization (MoSCoW Method) – Expanded Justifications]

What is the MoSCoW Method?

The **MoSCoW Method** is a prioritization technique used in software development and project management to classify requirements into four categories:

1. **Must Have:**
   These are **non-negotiable** requirements. The system **cannot function** without them. They form the **core functionality** of the product.
2. **Should Have:**
   Important, but not critical. These requirements **add significant value** and should be included if time and resources allow, but the system will still work without them.
3. **Could Have:**
   These are **nice-to-have** features that can improve user experience or open up future revenue or product expansion, but are **not necessary for launch**.
4. **Won't Have (for now):**
   These are **excluded from the current scope** of development, usually due to complexity, resource constraints, or strategic direction. They may be considered in future releases.

1. Must Have Requirements

These are **essential** for the car fault diagnosis mobile application to function:

- **Dashboard Light Recognition**
  This feature solves a real pain point for drivers: not understanding dashboard

symbols. Recognizing symbols like the check engine light or ABS warning enables users to take prompt action.

Business Value: High – builds immediate trust and usefulness.

Effort: ***Moderate*** **–** requires training a computer vision model with labeled icons.

- **Engine Sound Analysis**

  This is the app's **unique differentiator**, allowing users to diagnose faults through sound.

  Business Value: Extremely High – no common mobile apps offer this yet.

  Effort: ***High*** **–** involves collecting diverse audio data, training models, and building a robust recording interface.

2. Should Have Requirements

- **YouTube Video Tutorials**

  Providing video tutorials for DIY fixes enhances user **education** and **retention**.

  Business Value: Moderate to High – improves credibility, supports non-expert users.

  Effort: ***Low*** **–** API integration is straightforward.

3. Could Have Requirements

- **Mechanic Connection Platform**

  Connecting users to local, verified mechanics via the app could become a **monetizable feature**.

  Business Value: High – potential revenue stream.

  Effort*:* High – involves backend infrastructure, reviews, scheduling, and secure payments.

4. Won't Have (For Now)

- **OBD-II Scanner Integration**

  While common in professional tools, this feature **contradicts your app's goal** of being hardware-free and accessible to everyone.

Business Value*:* Niche – benefits only users with an OBD-II scanner.

Effort: Very High – requires Bluetooth/device compatibility, parsing proprietary data formats.

Priority	Requirement	Business Impact	Technical Effort
Must Have	Dashboard Light Recognition	Addresses most common driver confusion.	**Medium** – requires medium-sized CV model and training.
Must Have	Engine Sound Analysis	Critical for core app functionality and differentiator.	**High** – requires robust audio datasets, ML training, and UI integration.
Should Have	YouTube Videos	Increases user trust, Enhances credibility, user learning.	**Low** – quick integration via YouTube Data API.
Could Have	Mechanic Connection	Future revenue stream via consultations.	**High** – backend, payment, verification required.
Won't Have	OBD-II Integration	Contradicts hardware-free mission.	Very High – requires user-owned hardware and device compatibility layer.

4. Classification of Requirements

In software engineering, requirements are broadly categorized into **Functional Requirements** and **Non-Functional Requirements**.

4.1. Functional Requirements

Functional requirements are the **specific behaviors, features, or functions** that a system must perform or support. They describe **what the system should do**, from the perspective of both users and system components.

In other words, they define the **core operations** and **services** the software must provide to meet its objectives.

Key Functional Requirements:

1. **Dashboard Light Recognition:**

    - ✓ The app will be able to detect and interpret dashboard warning lights using the smartphone camera.
    - ✓ The app will be able to provide possible causes and urgency levels for each detected symbol.

2. **Engine Sound Analysis:**

    - ✓ The app will be able to record engine sounds using the smartphone microphone.
    - ✓ The app will be able to classify engine sounds using machine learning models.
    - ✓ The app will match recorded sounds to known fault patterns (e.g., knocking, squealing).

3. **Recommendation System:**

    - ✓ The app will suggest possible repair steps based on detected issues.
    - ✓ The app will provide links to relevant YouTube tutorials.

4. **Offline Mode:**

    - ✓ The app will function without an internet connection for basic light and sound recognition.
    - ✓ The app will store and recall the last five diagnostic results.

5. **User Interaction:**

    - ✓ The app will allow users to upload photos of dashboard symbols for analysis.
    - ✓ The app will provide confidence scores for each AI-based prediction.
    - ✓ The app will support user feedback on diagnosis accuracy.
    - ✓ The app will allow users to connect with verified mechanics.

4.2. Non-Functional Requirements

Non-functional requirements specify criteria that can be used to judge the operation of a system, rather than specific behaviors. They focus on how the system performs its functions, emphasizing quality attributes and operational constraints.

Key Non-Functional Requirements:

1. **Performance:**

    - ✓ The app will classify engine sounds within 5 seconds.
    - ✓ The app shall recognize dashboard symbols with at least 90% accuracy.

2. **Usability:**

    - ✓ The user interface will be simple, utilizing icons and minimal text.
    - ✓ Critical issues will trigger visual and voice alerts.

3. **Security & Privacy:**

    - ✓ User recordings shall be stored locally and deleted after 24 hours unless saved.
    - ✓ No recordings shall be shared without explicit user consent.

4. **Compatibility:**

    - ✓ The app shall support both Android and iOS operating systems.

5. **Maintainability:**

    - ✓ The app will be designed with modularity to facilitate updates and maintenance.
    - ✓ Machine learning models and tutorials shall be updateable via app updates.

6. **Reliability:**

    - ✓ The app will have an uptime of 99.9%.
    - ✓ The app will handle up to 1,000 concurrent users without performance degradation

7. **Portability:**

    - ✓ The app shall function correctly across the latest versions of iOS and Android devices

---

5. Validation with Stakeholders

5.1 Methods Used

- ✓ **Prototype Testing**: Low-fidelity and mid-fidelity versions presented
- ✓ **Surveys and Feedback Forms**: Used to measure satisfaction
- ✓ **Interviews**: Conducted with mechanics for common issues

5.2 Traceability Matrix

Requirement	Source	Priority	Validation Method
Dashboard Light Recognition	User Interview	High	Prototype Test
Engine Sound Classification	Expert Feedback	High	Model Testing
Tutorial Suggestion	Survey	Medium	Usability Testing
Offline Functionality	Stakeholder Request	High	Field Testing
Mechanic Linking	Owner Feedback	Medium	Interview