REPUBLIC OF CAMEROON

Peace-Work-Fatherland

MINISTRY OF HIGHER EDUCATION

UNIVERSITY OF BUEA

REPUBLIQUE DU CAMEROUN

Paix-Travail-Patrie

MINISTRE DE L'ENSEGNEMENT SUPERIEUR

UNIVERSITE DE BUEA

UNIVERSITY OF BUEA

FACULTY OF ENGINEERING AND TECHNOLOGY

TASK 6

DATABASE DESIGN AND IMPLEMENTATION

COURSE INSTRUCTOR: **Dr NKEMENI VALERY**

COURSE CODE/TITLE: **CEF 440/ INTERNET PROGRAMMING(J2EE) AND MOBILE PROGRAMMING.**

ACADEMIC YEAR 2024/2025

Table of Contents

Introduction

In the realm of modern mobile applications, the efficiency, reliability, and scalability of data management are paramount. This holds especially true for complex systems like a car fault diagnosis application, which deals with diverse and interconnected pieces of information, ranging from user profiles and vehicle specifics to intricate diagnostic codes, engine sound patterns, and extensive knowledge base articles. At the heart of such an application lies a robust database design and its subsequent implementation**.**

1. Data Elements

Data elements form the core of the system's information architecture. They define the schema for every piece of data captured, processed, and stored.

- **User Profile Data:** Fields include userID, name, email, hashed password, roles, and preferences. These elements ensure secure authentication and personalized experiences.
- **Vehicle Specifications:** Each Vehicle object contains make, model, year, VIN, engine type, and mileage. This structured data is crucial for accurate diagnostics and history tracking.
- **Diagnostic Results:** Comprising fault codes, descriptions, severity levels, timestamps, and recommended actions. Stored as nested objects within each Diagnosis record.
- **Sensor Inputs:** AudioSample elements capture raw audio byte streams, sampling rate, duration, and spectral metadata. DashboardImage elements include image resolution, format, and detected icon labels.
- **Metadata Attributes:** Timestamps, geolocation coordinates, device identifiers, and processing latency. These elements facilitate audit trails, performance analysis, and geo-specific service recommendations.
- **Configuration Settings:** Include threshold values for audio anomaly detection, image confidence cut-offs, and user notification preferences. Stored in a dedicated Config collection for dynamic updates.

- **Access Control Tags:** Labels indicating user roles, access rights, and sharing permissions, enabling granular security rules in Firestore.
- **Inter-Entity References:** Foreign key-like links such as userID→Vehicles, vehicleID→Diagnoses, diagnosisID→Media samples. These references maintain relational integrity in a NoSQL environment.
- **Data Lifecycle Management:** Elements defining retention periods, archival flags, and deletion markers to comply with data governance policies.

2. Conceptual Design

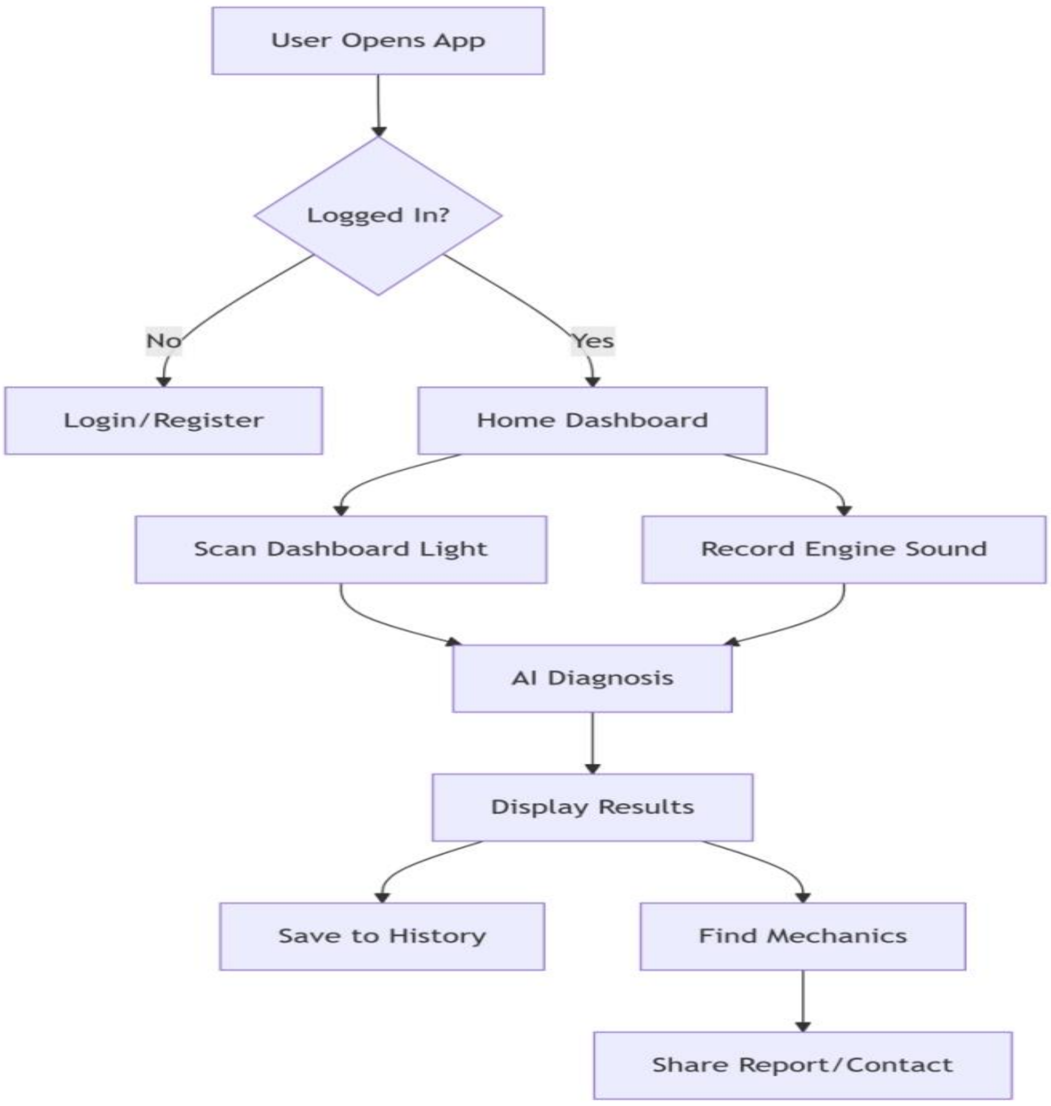
The conceptual design abstracts the system into high-level modules, defining responsibilities and data flows between them.

- **Presentation Layer:** Mobile frontend developed in React Native provides interfaces for capturing images, recording sounds, and displaying diagnostics.
- **Application Layer:** Contains controllers for handling input, validation services, and orchestrating ML model invocations either on-device or via cloud microservices.
- **Data Layer:** Firebase Firestore and Storage serve structured and unstructured data respectively. A service bus handles offline queueing and real-time synchronization.
- **AI Module:** Composed of two microservices—Image Recognition and Audio Analysis—hosted on cloud functions. They expose REST endpoints consumed by the Node.js API.
- **Notification Service:** Leverages Firebase Cloud Messaging to dispatch real-time alerts about urgent faults or maintenance reminders.

- **Security Layer:** Implements OAuth2 flows for user authentication, with JWTs for session management, and Firestore security rules enforcing role-based access.
- **Scalability Considerations:** Each module is containerized for horizontal scaling. Firestore indexes are designed to optimize query performance under high load.
- **Resilience Patterns:** Circuit breakers for ML service calls, retry policies for database operations, and local caching for disconnected scenarios.
- **Integration Points:** Third-party services like Google Maps API for mechanic locator and email/SMS gateways for sharing diagnostics.

The conceptual design focuses on the following,

- Core functionalities (what the app does)
- User flows (how users interact with the app)
- System architecture (how components interact)
- Key technologies (AI, APIs, databases)


User Opens App
Logged In?
No
Yes
Login/Register
Home Dashboard
Scan Dashboard Light
Record Engine Sound
AI Diagnosis
Display Results
Save to History
Find Mechanics
Share Report/Contact

3. ER Diagram

An Entity-Relationship (ER) Diagram is a high-level conceptual data model that visually represents the structure of a database. It illustrates the relationships between different entities (real-world objects or concepts) within a system. ER diagrams are composed of:

- **Entities:** Represented by rectangles, these are the main objects or concepts about which data is stored (e.g., User, Vehicle, Mechanic).
- **Attributes:** Represented by ovals, these are the properties or characteristics of an entity (e.g., a User has a Name, Email).
- **Relationships:** Represented by diamonds, these describe how entities are connected or associated with each other (e.g., a User *Registers* a Vehicle).
- **Cardinalities (or Multiplicities):** Numbers or symbols (e.g., 1, M, N) placed on the lines connecting entities to relationships, indicating the number of instances of one entity that can be associated with instances of another entity (e.g., One User can register Many Vehicles).

3.1. Importance of an ER Diagram

ER diagrams are crucial in database design for several reasons:

1. **Conceptual Clarity:** They provide a clear, easy-to-understand visual representation of the data structure, making it simpler for stakeholders (developers, business analysts, users) to comprehend the system's data requirements.
2. **Facilitates Communication:** They serve as a common language between technical and non-technical teams, reducing misunderstandings and ensuring everyone is on the same page regarding data organization.
3. **Foundation for Database Design:** An ER diagram acts as a blueprint for creating the physical database schema (tables, columns, keys) in relational database management systems (RDBMS) like MySQL, PostgreSQL, or SQL Server.
4. **Identifies Data Redundancy and Inconsistencies:** By visualizing relationships, designers can identify potential areas of redundant data or inconsistencies, allowing for better normalization and data integrity.
5. **Requirement Analysis:** The process of creating an ER diagram helps in thoroughly analyzing and defining data requirements for an application.

6. **Documentation:** It serves as vital documentation for the database structure, aiding in maintenance, future enhancements, and onboarding new team members.

3.2 Entities and Their Attributes in the AutoFix System

Here are the entities identified in the diagram, their attributes, and the reason for their inclusion:

1. **User**
    - **Reason for Entity:** Represents individuals who interact with the AutoFix system to register vehicles, request services, and view diagnostic information. We need to store their unique identity and personal details.
    - **Attributes:**
        - **UserID:** (Primary Key) - *Reason:* Uniquely identifies each user in the system.
        - **Name:** - *Reason:* Stores the user's full name for personalization and identification.
        - **Username:** - *Reason:* Provides a unique identifier for login purposes, often user-chosen.
        - **PasswordHash:** - *Reason:* Stores a securely hashed version of the user's password for authentication, without storing the plaintext password for security.
        - **Email:** - *Reason:* Stores the user's email address for communication, notifications, and potentially login.
        - **CreatedAt:** - *Reason:* Records the timestamp when the user account was created, useful for auditing and tracking user growth.
2. **Vehicle**
    - **Reason for Entity:** Represents the automobiles owned by users that require diagnostic services or maintenance. We need to track vehicle-specific information.
    - **Attributes:**
        - **VehicleID:** (Primary Key) - *Reason:* Uniquely identifies each vehicle in the system.
        - **Make:** - *Reason:* Stores the manufacturer of the vehicle (e.g., Toyota, BMW).
        - **Model:** - *Reason:* Stores the specific model of the vehicle (e.g., Camry, X5).

        - **Year:** - *Reason:* Stores the manufacturing year of the vehicle, relevant for diagnostics and parts.
        - **VIN (Vehicle Identification Number):** - *Reason:* A globally unique identifier for a vehicle, crucial for accurate identification, history, and recall information.
3. **Mechanic**
    - **Reason for Entity:** Represents the service providers (mechanics) available through the system. We need to store their professional details, contact information, and service capabilities.
    - **Attributes:**
        - **MechanicID:** (Primary Key) - *Reason:* Uniquely identifies each mechanic.
        - **Name:** - *Reason:* Stores the mechanic's or garage's name for display and identification.
        - **Phone:** - *Reason:* Primary contact number for users to reach the mechanic.
        - **Address:** - *Reason:* Physical location of the mechanic's shop or service area.
        - **Latitude, Longitude:** - *Reason:* Geographic coordinates for enabling map-based searches and proximity services.
        - **Rating:** - *Reason:* Represents the aggregated rating from users, indicating service quality.
        - **VerificationStatus:** - *Reason:* Indicates if the mechanic's credentials or business has been verified by the platform, building trust.
        - **Email (Optional):** - *Reason:* An alternative contact method for the mechanic.
        - **Website (Optional):** - *Reason:* A link to the mechanic's official website, if available.
        - **Specialties (Multi-valued):** - *Reason:* Lists the specific types of vehicles or repair areas the mechanic specializes in (e.g., "Engine Repair," "Brakes," "Electrical").
4. **LightCode (Warning Lights)**
    - **Reason for Entity:** Represents the standardized codes for vehicle warning lights. We need to store definitions for various warning signals.
    - **Attributes:**
        - **LightCode:** (Primary Key) - *Reason:* A unique identifier for each distinct warning light.

        - **SeverityLevel:** - *Reason:* Indicates the urgency or danger associated with the warning light (e.g., "Critical," "Moderate," "Informational").
        - **Description:** - *Reason:* Provides a detailed explanation of what the warning light signifies.

5. **Sound Pattern Reference**
    - **Reason for Entity:** Represents different distinctive sounds a vehicle might make, which can be indicators of problems. We need to categorize and describe these sounds.
    - **Attributes:**
        - **PatternID:** (Primary Key) - *Reason:* Uniquely identifies each distinct sound pattern.
        - **Component:** - *Reason:* Specifies which vehicle component is associated with the sound (e.g., "Engine," "Brakes," "Suspension").
        - **Description:** - *Reason:* Provides a detailed explanation of the sound and its characteristics.
6. **Video (Tutorial Videos)**
    - **Reason for Entity:** Represents instructional or tutorial videos related to vehicle issues or maintenance. We need to link to external video resources.
    - **Attributes:**
        - **VideoID:** (Primary Key) - *Reason:* Uniquely identifies each video.
        - **Title:** - *Reason:* The name of the video, for easy identification.
        - **URL:** - *Reason:* The direct link to where the video is hosted.
7. **Knowledge Base Entry**
    - **Reason for Entity:** Forms the core of the system's diagnostic and informational library, providing solutions and advice for vehicle problems. We need to store structured diagnostic content.
    - **Attributes:**
        - **EntryID:** (Primary Key) - *Reason:* Uniquely identifies each knowledge base article or entry.
        - **Cause:** - *Reason:* Describes the underlying reason for a set of symptoms or a problem.
        - **Symptoms:** - *Reason:* Details the observable signs or indications of a vehicle problem.
        - **Probability:** - *Reason:* A numerical value indicating the likelihood of a particular cause given specific symptoms.

        - **MaintenanceTips:** - *Reason:* Provides actionable advice or steps for resolving the issue or performing maintenance.

3.3 Relationships Between Entities

Relationships define how instances of one entity type are associated with instances of another. Associative entities (also known as junction or bridge tables) are used to resolve many-to-many relationships into one-to-many relationships in a relational database.

1. **Register (User - Vehicle)**
    - **Type:** One-to-Many (1:M)
    - **Description:** A User can **Register** multiple Vehicles, but each Vehicle is registered by only one User.
    - **Reason:** This relationship is needed to associate vehicles with their owners, enabling personalized vehicle management for each user.
2. **Diagnostic Session** (Associative Entity)
    - **Reason for Entity:** This entity captures the details of a specific diagnostic event or interaction for a vehicle. It's an associative entity because a User can have many Diagnostic Sessions, and a Vehicle can be involved in many Diagnostic Sessions.
    - **Attributes:**
        - **SessionID:** (Primary Key) - *Reason:* Uniquely identifies each diagnostic session.
        - **Confidence:** - *Reason:* Stores a metric of how confident the system/diagnosis is in its findings.
        - **InputType:** - *Reason:* Records how the diagnostic data was captured (e.g., "Manual Entry," "OBD Scanner," "Audio Analysis").
        - **UserID:** (Foreign Key to User) - *Reason:* Links the session to the user who initiated it.
        - **VehicleID:** (Foreign Key to Vehicle) - *Reason:* Links the session to the specific vehicle being diagnosed.
    - **Relationships:**
        - **Initiates (User - Diagnostic Session):** A User initiates Diagnostic Sessions.
        - **Undergoes (Vehicle - Diagnostic Session):** A Vehicle undergoes Diagnostic Sessions.
3. **Produce (Diagnostic Session - Consultation Request)**
    - **Type:** One-to-Many (1:M)

    - **Description:** A Diagnostic Session can **Produce** multiple Consultation Requests. Each Consultation Request is associated with one Diagnostic Session.
    - **Reason:** This relationship allows users to seek mechanic assistance directly from a specific diagnostic session's findings.
4. **Consultation Request** (Associative Entity)
    - **Reason for Entity:** This entity manages specific requests for mechanic consultation. It acts as an associative entity to record a user's request to a mechanic, potentially for a specific diagnostic session.
    - **Attributes:**
        - **RequestID:** (Primary Key) - *Reason:* Uniquely identifies each consultation request.
        - **Status:** - *Reason:* Tracks the current state of the request (e.g., "Pending," "Accepted," "Completed," "Declined").
        - **RequestedAt:** - *Reason:* Timestamp when the consultation request was made.
        - **MechanicID:** (Foreign Key to Mechanic) - *Reason:* Links the request to the specific mechanic.
        - **UserID:** (Foreign Key to User) - *Reason:* Links the request to the user who made it.
        - **SessionID:** (Foreign Key to Diagnostic Session) - *Reason:* Links the request to a specific diagnostic session (if applicable).
    - **Relationships:**
        - **Return (Mechanic - Consultation Request):** A Mechanic can handle many Consultation Requests.
        - **Return (User - Consultation Request):** A User can make many Consultation Requests.
5. **Diagnostic Session Sound Pattern** (Associative Entity)
    - **Reason for Entity:** This entity links specific Diagnostic Sessions to the Sound Pattern References that were identified or relevant during that session. It resolves a Many-to-Many relationship between Diagnostic Session and Sound Pattern Reference.
    - **Attributes:**
        - **SessionID:** (Foreign Key, part of Composite PK) - *Reason:* Links to the specific diagnostic session.
        - **PatternID:** (Foreign Key, part of Composite PK) - *Reason:* Links to the specific sound pattern detected.
        - **MatchProbability:** - *Reason:* A numerical value indicating how likely the detected sound matches this reference pattern.

- **Relationships:**
    - **Links Diagnostic Session to Sound Pattern Reference (Many-to-Many)**

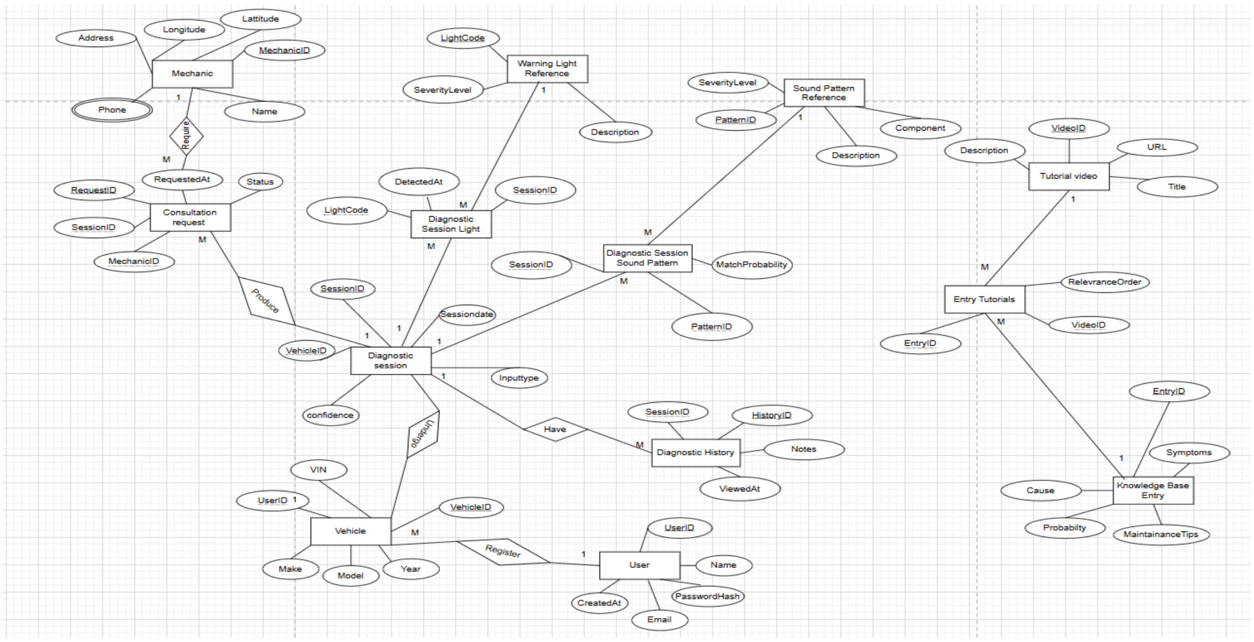
6. **Diagnostic Session Light** (Associative Entity)
    - **Reason for Entity:** This entity links specific Diagnostic Sessions to the LightCode (warning lights) that were detected during that session. It resolves a Many-to-Many relationship between Diagnostic Session and LightCode.
    - **Attributes:**
        - **SessionID:** (Foreign Key, part of Composite PK) - *Reason:* Links to the specific diagnostic session.
        - **LightCode:** (Foreign Key, part of Composite PK) - *Reason:* Links to the specific warning light detected.
        - **DetectedAt:** - *Reason:* Timestamp when the warning light was detected in the session.
    - **Relationships:**
        - **Links Diagnostic Session to LightCode (Many-to-Many)**
7. **Diagnostic History** (Associative Entity)
    - **Reason for Entity:** This entity stores historical records or notes related to individual Diagnostic Sessions.
    - **Attributes:**
        - **HistoryID:** (Primary Key) - *Reason:* Uniquely identifies each history entry.
        - **SessionID:** (Foreign Key to Diagnostic Session) - *Reason:* Links the history entry to its corresponding diagnostic session.
        - **Notes:** - *Reason:* Stores free-form text notes about the session's history or outcomes.
        - **ViewedAt:** - *Reason:* Timestamp indicating when this history entry was last viewed.
    - **Relationships:**
        - **Have (Diagnostic Session - Diagnostic History):** A Diagnostic Session can have many Diagnostic History entries, but each history entry belongs to one Diagnostic Session.
8. **Entry Tutorials** (Associative Entity)
    - **Reason for Entity:** This entity connects Knowledge Base Entries to relevant Videos that serve as tutorials. It resolves a Many-to-Many relationship between Knowledge Base Entry and Video.
    - **Attributes:**
        - **EntryID:** (Foreign Key, part of Composite PK) - *Reason:* Links to the specific knowledge base entry.

        - **VideoID:** (Foreign Key, part of Composite PK) - *Reason:* Links to the specific tutorial video.
        - **RelevanceOrder:** - *Reason:* Specifies the order in which videos should be presented for a particular knowledge base entry, indicating their importance or sequence.
    - **Relationships:**
        - **Links Knowledge Base Entry to Video (Many-to-Many)**

3.4 Summary and Importance of the Design

This ER Diagram provides a robust foundation for the AutoFix application's database. By clearly defining entities, their attributes, and relationships, it ensures:

- **Data Integrity:** Relationships with foreign keys prevent orphaned data and ensure consistency.
- **Scalability:** The structured approach allows for easy expansion as more features or data types are introduced.
- **Efficient Data Retrieval:** A well-designed schema facilitates optimized queries for fetching relevant information quickly (e.g., finding all mechanics in a region, or all diagnostic sessions for a particular vehicle).
- **Reduced Redundancy:** Associative entities correctly normalize many-to-many relationships, preventing duplicate data storage and improving data quality.



4. Database Implementation

This report details the database implementation for the AutoFix Car system, leveraging Firebase Firestore. It translates the conceptual Entity-Relationship (ER) Diagram into a practical NoSQL database structure, outlining the collections (equivalent to tables in a relational database), their key fields (columns), and their importance to the system's functionality.

4.1 Database Choice: Firebase Firestore

Firebase Firestore is a NoSQL, document-oriented database. Instead of traditional "tables" and "rows," Firestore organizes data into collections of documents. Each document can contain various key-value pairs (fields) and can also contain subcollections.

Why Firestore for AutoFix Car System?

- **Scalability:** Firestore automatically scales to handle large user bases and massive amounts of data, crucial for the requirement of 1M+ vehicle profiles and 100,000 concurrent users.
- **Real-time Synchronization:** Its real-time capabilities allow for immediate updates across connected clients, beneficial for features like mechanic consultation status or diagnostic data updates.
- **Offline Support:** Native SDKs provide robust offline data persistence and synchronization, ensuring core functionalities work even without an internet connection.
- **Flexibility (NoSQL):** The schema-less nature of Firestore allows for agile development and easy adaptation to evolving data requirements without rigid migrations.
- **Integrated Ecosystem:** Seamlessly integrates with other Firebase services like Authentication, Storage, and Cloud Functions, streamlining backend development.

4.2 Database Structure: Collections and Fields

Based on the above ER Diagram, here's how the entities and their attributes translate into Firestore collections and documents:

4.2.1. Users Collection

- **Corresponds to ER Diagram Entity:** User

- **Purpose:** Stores information about every user registered in the AutoFix system.
- **Importance:** Essential for user authentication, personalization (linking vehicles and history to users), and communication.
- **Key Fields (Document Attributes):**
    - userId (Document ID / Field, if also stored as a field): Unique identifier for the user. **Importance:** Primary key for user identification and lookup.
    - name: User's full name. **Importance:** Personalization and display.
    - username: User's chosen username. **Importance:** Alternative login identifier.
    - email: User's email address. **Importance:** Communication, notifications, and login.
    - createdAt: Timestamp of user creation. **Importance:** Auditing and tracking user growth.
    - passwordHash: Stored securely. **Importance:** Secure authentication.

4.2.2. Vehicles Collection

- **Corresponds to ER Diagram Entity:** Vehicle
- **Purpose:** Stores details about each vehicle registered by users.
- **Importance:** Fundamental for associating diagnostic sessions and history with specific cars.
- **Key Fields:**
    - vehicleId (Document ID / Field, if also stored as a field): Unique identifier for the vehicle. **Importance:** Primary key for vehicle identification.
    - userId: (Foreign Key) ID of the User who owns this vehicle. **Importance:** Establishes the 1:M relationship between User and Vehicle, allowing retrieval of all vehicles for a user.
    - make: Vehicle manufacturer. **Importance:** Categorization and display.
    - model: Vehicle model. **Importance:** Specific identification.
    - year: Manufacturing year. **Importance:** Relevance for parts, diagnostics, and knowledge base.
    - vin: Vehicle Identification Number. **Importance:** Global unique identifier, crucial for accurate identification.

4.2.3. Mechanics Collection

- **Corresponds to ER Diagram Entity:** Mechanic
- **Purpose:** Stores information about mechanics available for consultation.
- **Importance:** Enables users to find and connect with qualified service providers.
- **Key Fields:**
    - mechanicId (Document ID / Field): Unique identifier for the mechanic. **Importance:** Primary key for mechanic identification.
    - name: Mechanic's or shop's name. **Importance:** Identification and display.
    - phone: Contact number. **Importance:** Direct communication.
    - address: Physical address. **Importance:** Location-based searching.
    - latitude, longitude: Geographic coordinates. **Importance:** Enables proximity searches and map integration.
    - rating: Aggregated rating. **Importance:** Indicates service quality.
    - verificationStatus: Status of platform verification. **Importance:** Builds trust and credibility.
    - email (Optional): Alternative contact. **Importance:** Additional communication channel.
    - website (Optional): Link to personal/shop website. **Importance:** Provides more information to users.
    - specialties (Array of Strings or Subcollection specialties): Areas of expertise. **Importance:** Enables specialized searches for users.

4.2.4. LightCodes Collection

- **Corresponds to ER Diagram Entity:** LightCode
- **Purpose:** Stores definitions and details for various vehicle dashboard warning lights.
- **Importance:** Provides the foundational data for the "Dashboard Warning Light Recognition" feature.
- **Key Fields:**
    - lightCode (Document ID / Field): Unique code (e.g., "P0301", "Check Engine"). **Importance:** Primary key for light code identification.
    - severityLevel: Level of urgency/danger. **Importance:** Helps users prioritize actions.
    - description: Detailed explanation of the light. **Importance:** Informs users about the problem.

4.2.5. SoundPatternReferences Collection

- **Corresponds to ER Diagram Entity:** Sound Pattern Reference
- **Purpose:** Stores descriptions and associations for different vehicle sound patterns.
- **Importance:** Provides the reference data for the "Engine Sound Analysis" feature.
- **Key Fields:**
  - patternId (Document ID / Field): Unique identifier for the sound pattern. **Importance:** Primary key for sound pattern identification.
  - component: Vehicle component associated with the sound. **Importance:** Helps narrow down the source of the issue.
  - description: Detailed description of the sound. **Importance:** Explains the characteristics of the pattern.

4.2.6. Videos Collection

- **Corresponds to ER Diagram Entity:** Video
- **Purpose:** Stores links and titles for tutorial videos.
- **Importance:** Provides valuable visual resources for DIY repairs and maintenance.
- **Key Fields:**
  - videoId (Document ID / Field): Unique identifier for the video. **Importance:** Primary key for video identification.
  - title: Title of the video. **Importance:** Helps users identify content.
  - url: Direct link to the video content. **Importance:** Enables access to the tutorial.

4.2.7. KnowledgeBaseEntries Collection

- **Corresponds to ER Diagram Entity:** Knowledge Base Entry
- **Purpose:** Stores diagnostic information, causes, symptoms, and maintenance tips.
- **Importance:** Forms the core "intelligence" of the diagnosis system, guiding users through troubleshooting and providing solutions.
- **Key Fields:**
  - entryId (Document ID / Field): Unique identifier for the knowledge base entry. **Importance:** Primary key for entry identification.
  - cause: Description of the problem's cause. **Importance:** Helps users understand the root issue.
  - symptoms: Description of the symptoms. **Importance:** Allows matching user input to potential problems.

- probability: Likelihood of cause given symptoms. **Importance:** Helps prioritize potential diagnoses.
- maintenanceTips: Actionable advice. **Importance:** Guides users on how to resolve or manage the issue.

4.2.8. DiagnosticSessions Collection

- **Corresponds to ER Diagram Entity:** Diagnostic Session (Associative Entity)
- **Purpose:** Records each diagnostic event initiated by a user for a specific vehicle.
- **Importance:** Central hub for linking various diagnostic inputs, results, and subsequent actions (like consultation requests).
- **Key Fields:**
  - sessionId (Document ID / Field): Unique identifier for the session. **Importance:** Primary key for session identification.
  - userId: (Foreign Key) ID of the User who initiated the session. **Importance:** Links session to user.
  - vehicleId: (Foreign Key) ID of the Vehicle involved. **Importance:** Links session to vehicle.
  - confidence: Confidence level of the diagnosis. **Importance:** Indicates reliability of the system's assessment.
  - inputType: How data was gathered (e.g., "Manual", "OBD Scan", "Audio"). **Importance:** Contextualizes the diagnostic input.
  - createdAt: Timestamp of session creation. **Importance:** Tracks when diagnosis occurred.

4.2.9. Associative Collections (for Many-to-Many Relationships)

Firestore handles many-to-many relationships by creating separate collections that contain references (document IDs) to documents in other collections.

- **DiagnosticSessionLights Collection**
  - **Corresponds to ER Diagram:** Association between Diagnostic Session and LightCode.
  - **Purpose:** Links specific DiagnosticSessions to the LightCodes detected during them.
  - **Importance:** Records all warning lights present in a specific diagnostic event.
  - **Key Fields:**

        - sessionId: (Foreign Key) ID of the DiagnosticSession.
        - lightCode: (Foreign Key) ID of the LightCode detected.
        - detectedAt: Timestamp of detection. **Importance:** Provides context on when the light appeared in the session.
        - (Document ID for this collection would typically be auto-generated or a composite of sessionId and lightCode if a custom one is needed).

- **DiagnosticSessionSounds Collection**
    - **Corresponds to ER Diagram:** Association between Diagnostic Session and Sound Pattern Reference.
    - **Purpose:** Links DiagnosticSessions to the SoundPatternReferences identified.
    - **Importance:** Records all sound patterns identified in a specific diagnostic event.
    - **Key Fields:**
        - sessionId: (Foreign Key) ID of the DiagnosticSession.
        - patternId: (Foreign Key) ID of the SoundPatternReference.
        - matchProbability: Likelihood of match. **Importance:** Quantifies the relevance of the sound pattern.
        - (Document ID for this collection would typically be auto-generated or a composite).
- **ConsultationRequests Collection**
    - **Corresponds to ER Diagram:** Consultation Request (Associative Entity)
    - **Purpose:** Records a user's request for consultation from a mechanic, linked to a specific diagnostic session.
    - **Importance:** Facilitates communication and scheduling between users and mechanics.
    - **Key Fields:**
        - requestId (Document ID / Field): Unique ID for the request.
        - userId: (Foreign Key) ID of the requesting User.
        - mechanicId: (Foreign Key) ID of the Mechanic requested.
        - sessionId: (Foreign Key, Optional) ID of the related DiagnosticSession. **Importance:** Links the request to the diagnostic context.
        - status: Current status (e.g., "Pending", "Accepted", "Completed"). **Importance:** Tracks the lifecycle of the request.
        - requestedAt: Timestamp of the request. **Importance:** For tracking and display.

- **EntryTutorials Collection**
    - **Corresponds to ER Diagram:** Association between Knowledge Base Entry and Video.
    - **Purpose:** Links KnowledgeBaseEntries to relevant Videos.
    - **Importance:** Organizes and presents tutorial videos effectively within the knowledge base.
    - **Key Fields:**
        - entryId: (Foreign Key) ID of the KnowledgeBaseEntry.
        - videoId: (Foreign Key) ID of the Video.
        - relevanceOrder: Order of importance. **Importance:** Guides the display sequence of videos.
        - (Document ID for this collection would typically be auto-generated or a composite).
- **DiagnosticHistory Collection**
    - **Corresponds to ER Diagram:** Diagnostic History (Associative Entity with Diagnostic Session)
    - **Purpose:** Stores historical notes or specific views related to a diagnostic session.
    - **Importance:** Allows users to add personal notes to their diagnostic records.
    - **Key Fields:**
        - historyId (Document ID / Field): Unique ID for the history entry.
        - sessionId: (Foreign Key) ID of the DiagnosticSession. **Importance:** Links to the specific diagnostic event.
        - notes: Free-form text notes. **Importance:** Captures user-specific insights or follow-ups.
        - viewedAt: Timestamp when the entry was viewed. **Importance:** For tracking user interaction.

4.3 Importance of a Well-Defined Database Structure

A clear and well-defined database structure, as outlined above, is paramount for the AutoFix system's success because it directly impacts:

- **Data Integrity:** By defining clear relationships and foreign keys (even if implicit in NoSQL by referencing document IDs), it helps maintain consistency and prevents orphaned or contradictory data.

- **Query Efficiency:** Proper collection design and thoughtful indexing in Firestore (which is often handled automatically for simple queries, but can be manually optimized for complex ones) enable the backend to retrieve data quickly, meeting performance requirements.
- **Maintainability and Extensibility:** A logical structure makes it easier for developers to understand the data model, simplifying future additions of features or modifications to existing ones.
- **Security:** Firestore's security rules rely on a well-structured database to define precise access controls for different users and roles.
- **Application Logic Simplicity:** A clear database structure simplifies the Node.js backend's code, as it knows exactly where to store and retrieve specific pieces of information.

5. Backend Implementation

The backend, also known as the "server-side" of an application, refers to the components that are not directly accessible by the user. It handles the logic, data storage, and communication between the mobile application (frontend) and external services. For the AutoFix system, the backend will be the central hub for processing diagnostic requests, managing user and mechanic data, serving knowledge base content, and enabling communication features.

5.1. Purpose and Use in the System

The backend's primary purposes in the AutoFix system are to:

- **Data Management:** Store, retrieve, and manage all persistent data, including user profiles, vehicle details, diagnostic history, mechanic information, and the knowledge base.
- **Business Logic Execution:** Implement the core business rules and algorithms, such as processing diagnostic results from the mobile app, managing user authentication, and handling consultation requests.
- **API Provision:** Expose a set of APIs (Application Programming Interfaces) that the mobile application can consume to interact with the system's data and functionality.
- **Scalability and Performance:** Ensure the system can handle a large number of concurrent users and data, maintaining good response times as per non-functional requirements.
- **Security:** Protect sensitive user data, implement secure authentication, and authorize access to resources.
- **Integration:** Facilitate integration with third-party services like map APIs, notification systems, and potentially external diagnostic tools.

5.2 Backend Implementation Details (Node.js & Firebase)

We used Node.js for building the backend application server and Firebase as our primary database solution.

5.2.1 Node.js Backend Server

Framework: Express.js (a popular, minimalist web framework for Node.js) **Why Node.js with Express.js?**

- **Scalability (Non-Functional Requirement):** Node.js's event-driven, non-blocking I/O model makes it highly efficient for handling concurrent requests, supporting the requirement for 100,000 concurrent users.
- **Performance (Non-Functional Requirement):** Its speed and efficiency are crucial for meeting response time requirements for dashboard light recognition (3 seconds), sound analysis (10 seconds), and UI interactions (0.5 seconds).
- **JSON Handling:** Native support for JSON, which is ideal for RESTful API communication with a mobile frontend and NoSQL databases like Firestore.
- **Vast Ecosystem:** A rich ecosystem of npm packages simplifies development tasks (e.g., authentication, data validation).
- **Full-Stack JavaScript:** Allows for a unified language across frontend (Flutter often uses Dart, but Node.js for backend can streamline developer understanding) and backend, potentially accelerating development.

Core Components of the Node.js Backend:

1. **API Endpoints:** RESTful APIs will be designed to expose backend functionalities.
2. **Authentication & Authorization Middleware:** To secure API endpoints.
3. **Controllers:** Handle incoming requests, interact with services, and send responses.
4. **Services/Logic Layer:** Encapsulate business logic and interactions with the database.
5. **Firebase Admin SDK Integration:** For secure server-side interaction with Firebase services.

5.2.2 Firebase Database

Database Type: Firestore (NoSQL, document-oriented database) **Why Firebase (Firestore)?**

- **Scalability (Non-Functional Requirement):** Firestore is designed for massive scale, accommodating 1M+ vehicle profiles and supporting concurrent users.
- **Real-time Capabilities:** While not extensively detailed in the SRS, real-time updates for things like new mechanic availability or diagnostic status can be easily implemented if needed, complementing "Real-time alerts while driving".

- **Backend as a Service (BaaS):** Firebase provides a comprehensive suite of services (Authentication, Cloud Functions, Storage, Hosting) which can significantly reduce backend development time and infrastructure management.
- **Offline Support (Non-Functional Requirement):** Firestore offers robust offline capabilities that can sync data when the connection resumes, directly supporting the "synchronize offline activity when connection resumes" requirement.
- **Security Rules:** Granular, declarative security rules for data access control directly from the client, reducing server-side validation overhead for basic operations.

Firebase Services Utilized:

- **Firestore:** For storing structured data.
- **Firebase Authentication:** For user authentication.
- **Firebase Storage:** For storing static assets like images (e.g., dashboard light images, mechanic profile pictures).
- **Firebase Cloud Functions (Optional but Recommended):** For server-side logic triggered by database events or HTTP requests, especially for complex or sensitive operations not handled directly by Node.js server.

5.3. Implementation of Key Functional Requirements in the Backend

Here's how the backend, using Node.js and Firebase, would implement the core functional requirements from your SRS:

5.3.1 Core Diagnostic Features

- **Dashboard Warning Light Recognition:**
  - **Backend Flow:** The mobile app uploads an image of the dashboard light to Firebase Storage. A Node.js API endpoint (e.g., /diagnose/light) receives the storage URL.
  - **Processing:** The Node.js backend (or potentially a Cloud Function triggered by the image upload) would then send this image to a third-party image recognition API (e.g., Google Cloud Vision AI, custom ML model hosted separately) that specializes in vehicle dashboard symbols.
  - **Database Interaction:** Once the light is identified, the backend queries Firestore's LightCodes collection (corresponding to your

    LightCode entity in the ERD) to retrieve the detailed explanation and severity level.
  - **Response:** The Node.js server sends back the diagnosis, explanation, and severity to the mobile app.
- **Engine Sound Analysis:**
  - **Backend Flow:** The mobile app records and uploads the engine sound audio file to Firebase Storage. A Node.js API endpoint (e.g., /diagnose/sound) receives the storage URL.
  - **Processing:** The Node.js backend sends the audio file to a specialized audio analysis service or a deployed ML model (e.g., Python-based service invoked by Node.js) that can analyze engine sounds and match them to known fault patterns.
  - **Database Interaction:** Based on the identified sound pattern, the backend queries Firestore's SoundPatternReferences collection to get component and description details.
  - **Response:** The analysis results, including identified component and potential issues, are sent back to the mobile app.
- **Problem Diagnosis:**
  - **Backend Flow:** After light and/or sound analysis (or manual input from the user), the mobile app sends the combined input data to a Node.js API endpoint (e.g., /diagnose/problem).
  - **Logic:** The Node.js backend implements the diagnostic logic. This could involve:
    - Comparing inputs against rules in a KnowledgeBaseEntries collection (from your ERD) that link symptoms to causes with probability ratings.
    - Applying a simple expert system or decision tree logic.
    - For complex cases, invoking a more sophisticated ML model via an external API.
  - **Database Interaction:**
    - Retrieves KnowledgeBaseEntry data for symptom-to-cause mapping, probability ratings, and maintenance tips.
    - Stores the DiagnosticSession details (confidence, input type, linked User/Vehicle, light codes, sound patterns) in Firestore.
  - **Response:** The backend provides a preliminary diagnosis, estimated repair urgency, linked causes, and maintenance tips to the mobile app.

5.3.2. User Management & Vehicle Information

- **User Profile Management:**

  - **Authentication (Firebase Auth):** Firebase Authentication will handle user registration (email/password, phone, Google Sign-In, etc.) and login securely. Node.js backend will integrate with Firebase Admin SDK to verify user tokens for authenticated requests.
  - **User Data (Firestore):** A Users collection in Firestore will store UserID, Name, Username, Email, CreatedAt.
  - **Vehicle Details (Firestore):** A Vehicles sub-collection under each User document (or a top-level Vehicles collection with UserID as a field) will store VehicleID, Make, Model, Year, VIN. Node.js API endpoints (/user/vehicles, /user/vehicles/:id) will manage these.
- **Vehicle History:**
  - **Storage (Firestore):** The DiagnosticSession collection (as discussed above) serves as the primary history. A DiagnosticHistory sub-collection under DiagnosticSession can store additional Notes and ViewedAt timestamps.
  - **Retrieval:** Node.js API endpoints (e.g., /vehicles/:id/history) will query Firestore to retrieve a vehicle's diagnostic history, maintenance records, and generate maintenance reminders (logic for reminders would be in Node.js, possibly triggered by scheduled Cloud Functions).

5.3.3. Support Features

- **Mechanic Connectivity:**
  - **Directory (Firestore):** A Mechanics collection in Firestore will store MechanicID, Name, Phone, Address, Latitude, Longitude, Rating, VerificationStatus, Email, Website, and Specialties.
  - **Search/Filter API (Node.js):** Node.js API endpoints (e.g., /mechanics, /mechanics?region=X&specialty=Y) will handle queries to Firestore for nearby mechanics, leveraging location data (Latitude/Longitude) for proximity searches.
  - **Sharing Results:** The Node.js backend might generate shareable links or formats for diagnostic results, or simply facilitate the transfer of data directly to a mechanic's account within the system (if mechanics also have accounts).
  - **Direct Consultation (Optional):** If implemented, this could involve creating a ConsultationRequests collection in Firestore, managed by Node.js APIs, allowing users to send requests and mechanics to respond.
- **Knowledge Base:**

    - **Storage (Firestore):** KnowledgeBaseEntries collection for EntryID, Cause, Symptoms, Probability, MaintenanceTips.
    - **Video Integration (Firestore & External):** Videos collection for VideoID, Title, URL. The EntryTutorials collection (associative entity) links KnowledgeBaseEntries to Videos.
    - **Search/Retrieval API (Node.js):** Node.js API endpoints (e.g., /knowledgebase/search?query=X, /knowledgebase/entry/:id) will query these collections in Firestore to provide searchable content, DIY guides, and preventative maintenance recommendations.

5.3.4. Offline Functionality & Synchronization

- **Offline Data (Firestore's Native Offline Support):** Firestore's SDK automatically handles offline data persistence and synchronization when the connection resumes, which is a major advantage for "Essential diagnostic features shall work without internet connection".
- **Synchronization Logic (Firebase SDK & Cloud Functions):** The Firebase SDK in the mobile app manages local data and pushes changes to Firestore when online. For more complex server-side processing of offline data, Firebase Cloud Functions can be triggered by new or updated documents in Firestore (e.g., processing a diagnostic session initiated offline once it syncs).
- **Critical Reference Data Storage:** The backend (Node.js server and Cloud Functions) ensures that the LightCodes, SoundPatternReferences, and core KnowledgeBaseEntries are efficiently structured and potentially cached for quick access by the mobile app's offline capabilities.

5.4 Non-Functional Requirements Implementation

- **Performance:** Node.js's async nature and Firestore's scalability address response time and user load. Database indexing in Firestore will be critical for query performance.
- **Accuracy:** While the backend facilitates data flow, the accuracy (>95% for light, >85% for sound) primarily depends on the quality of the underlying ML models and data. The backend ensures correct data input to these models and proper logging of confidence levels.
- **Resource Utilization:** Node.js applications are generally lightweight in terms of memory. Firestore handles database storage efficiently. Firebase Storage manages image/audio files effectively.

- **Reliability:** Firestore is a highly available service (99.99% uptime). Node.js backend can be deployed with redundancy (e.g., multiple instances) to ensure high availability (99.5% availability requirement). Error handling, input validation, and data backups in Firebase are crucial for fault tolerance and data protection.
- **Security:**
    - **Data Protection:** Firebase's built-in encryption for data at rest and in transit covers a significant part. Node.js backend will enforce data validation and sanitization.
    - **Authentication:** Firebase Authentication for secure user management.
    - **Authorization:** Firebase Security Rules for client-side data access control, complemented by Node.js middleware for complex or sensitive server-side logic and API access control.
- **Scalability:** Node.js clusters/containerization (e.g., Docker/Kubernetes) combined with Firebase's inherent scalability will ensure the system can handle a large user base and growing data volumes.
- **Legal and Compliance:** The backend would log user consents, manage disclaimers (e.g., ensuring they are presented to users via API calls), and store user data in a way that respects privacy regulations.

6. Database Connection to the Backend

Database connectivity refers to the process by which a backend application establishes and maintains communication with a database system. This connection allows the backend to perform operations such as storing, retrieving, updating, and deleting data (CRUD operations) that are essential for the application's functionality. For the AutoFix system, this means the Node.js backend interacting seamlessly with the Firebase database.

6.1. Use of Database Connectivity in the AutoFix System

The connection between the Node.js backend and the Firebase database is fundamental to nearly every functional requirement of the AutoFix mobile application. It serves the following critical uses:

- **User and Vehicle Management:**
    - When a new user registers, the Node.js backend uses the connection to Firebase Authentication to create the user account and then stores

    user profile details (e.g., UserID, Name, Email, CreatedAt) in a Firestore Users collection.
    - Users can add multiple vehicles. The backend connects to Firestore to store VehicleID, Make, Model, Year, VIN associated with the user.
    - Retrieval of user and vehicle data for profile display or specific vehicle selection is handled via this connection.
- **Diagnostic Data Management:**
    - For dashboard warning light recognition, the mobile app sends image URLs to the Node.js backend. The backend uses the connection to Firebase Storage to access these images (if needed for further processing) and then queries a Firestore LightCodes collection to retrieve explanations and severity levels based on the recognized light code.
    - Similarly, for engine sound analysis, the backend accesses audio files from Firebase Storage and queries a Firestore SoundPatternReferences collection for relevant sound patterns.
    - All diagnostic sessions, including their Confidence, InputType, and links to User and Vehicle, are stored in a Firestore DiagnosticSessions collection. Associations with LightCode and Sound Pattern Reference are managed via Diagnostic Session Light and Diagnostic Session Sound Pattern collections.
- **Mechanic Information and Consultation:**
    - Mechanic profiles (containing MechanicID, Name, Phone, Address, Latitude, Longitude, Rating, VerificationStatus, etc.) are stored in a Mechanics collection in Firestore. The backend's connection enables searching and filtering these mechanics by location, specialty, or rating.
    - Consultation Request details, including RequestID, Status, and links to SessionID, MechanicID, and UserID, are managed in Firestore. The backend facilitates the creation, update, and retrieval of these requests.
- **Knowledge Base and Tutorials:**
    - The rich content of the knowledge base, including EntryID, Cause, Symptoms, Probability, and MaintenanceTips, is stored in a KnowledgeBaseEntries collection in Firestore.
    - Video details (VideoID, Title, URL) are stored in a Videos collection, and their association with knowledge base entries is managed via an Entry Tutorials collection in Firestore. The backend connects to these collections to serve relevant repair guides and videos to users.
- **Diagnostic History:**

    - Records of past diagnostic sessions and any associated Notes or ViewedAt timestamps are stored in a Diagnostic History collection (or similar structure) in Firestore, linked to specific Diagnostic Session entries. The backend provides APIs for users to view their complete diagnostic history.
- **Offline Functionality and Synchronization:**
    - Firebase Firestore inherently provides robust offline capabilities. The Node.js backend serves as the authoritative source for data. When the mobile app is online, the Firebase SDK (client-side) automatically synchronizes local changes with Firestore. The backend's consistent data model in Firestore is crucial for this seamless synchronization.

6.2 Importance of Database Connectivity

The robust and efficient connection between the Node.js backend and the Firebase database is paramount for the AutoFix system's success, directly impacting its non-functional requirements:

1. **Data Persistence and Integrity:**
    - **Importance:** Ensures that all user data, vehicle details, diagnostic records, and knowledge base content are reliably stored and retrieved even after the application is closed. It maintains the consistency and accuracy of data across the system.
    - **How Node.js & Firebase achieve this:** Firestore, as a NoSQL database, offers strong consistency models for its data. The Node.js backend, through the Firebase Admin SDK, ensures that data is written and read correctly according to the defined schema and relationships (as per the ERD).
2. **Scalability:**
    - **Importance:** The system must be able to handle a growing number of users (up to 100,000 concurrent users) and a vast amount of data (1M+ vehicle profiles) without performance degradation.
    - **How Node.js & Firebase achieve this:** Node.js's non-blocking I/O model makes it highly scalable for concurrent connections. Firebase Firestore is a managed, serverless database that automatically scales to handle large volumes of reads and writes without requiring manual server provisioning from the development team. This combination ensures the application can grow with its user base.
3. **Performance and Response Time:**

    - **Importance:** Users expect quick responses, especially for critical features like dashboard light recognition (within 3 seconds) and sound analysis (within 10 seconds).
    - **How Node.js & Firebase achieve this:** Node.js is known for its speed in I/O-bound operations, making it efficient for handling API requests and database interactions. Firestore provides low-latency data access globally. Proper indexing in Firestore and optimized queries from the Node.js backend are key to meeting these performance targets.
4. **Data Security:**
    - **Importance:** Protecting sensitive user data (like email and potentially location data) and ensuring secure authentication are critical for user trust and compliance.
    - **How Node.js & Firebase achieve this:** Firebase Authentication handles user credentials securely. Firestore provides powerful, rule-based security mechanisms that allow fine-grained control over data access directly from the client, while the Node.js backend with the Firebase Admin SDK provides a secure server-side path for more privileged operations. Data is encrypted at rest and in transit by Firebase.
5. **Offline Functionality:**
    - **Importance:** The application needs to work without an internet connection for essential diagnostic features, with data synchronizing once online.
    - **How Node.js & Firebase achieve this:** Firebase Firestore's SDK comes with built-in offline persistence and synchronization. The backend (Firebase itself) ensures that the data model supports this, and any server-side logic (e.g., via Cloud Functions) can process data that syncs after an offline period.
6. **Development Efficiency:**
    - **Importance:** Accelerating the development process to deliver features efficiently.
    - **How Node.js & Firebase achieve this:** Firebase is a Backend as a Service (BaaS) that significantly reduces the need to build and maintain complex backend infrastructure. Node.js with Express.js offers a rapid development environment. This allows the team to focus more on core application logic rather than database management.

Conclusion

The comprehensive exploration of database design and implementation for the Auto Fix Car application underscores its pivotal role in the system's success. This report has detailed the conceptual data model, its translation into a NoSQL database structure, and the critical connectivity layer that binds the application's components.

In conclusion, the meticulous database design, grounded in the Entity-Relationship (ER) Diagram, has provided a clear, logical blueprint for organizing diverse data elements. This systematic approach ensures data integrity, minimizes redundancy, and forms a solid foundation for all application functionalities from managing user and vehicle profiles to handling complex diagnostic data, maintaining a rich knowledge base, and facilitating mechanic consultations.