**System Structure Report: Car Fault Check Startup**

**1. Introduction**

Welcome to Smart-car, dedicated to streamlining car diagnostics by providing a user-friendly platform for checking faults using OBD technology. This report outlines the comprehensive system structure for the development of our website and mobile app**.**

**2. System Overview**

The central components of our system include an intuitive website and a mobile application. These platforms will seamlessly integrate with OBD devices to conduct efficient and precise car fault checks**.**

**3. Website and Mobile App Architecture**

3.1 Technology Stack

Front-end:

Website: React.js

Mobile App: React Native

Back-end: by aws or google cloud

Node.js for server-side logic

Express.js as the web application framework

MongoDB as the NoSQL database for flexible data storage

3.2 Interaction

The website and mobile app will communicate with our backend server, which will interface with the OBD system through standardized protocols and APIs (GraphQl).

**4. User Authentication and Authorization**

User authentication will be implemented using JWT tokens (**JSON Web Token** ) to ensure secure access to the system. Different user roles, such as admin, mechanic, and car owner, will have distinct permissions within the platform.

**5. OBD Integration**

5.1 Communication Protocols

We will leverage industry-standard communication protocols like OBD-II and relevant APIs to establish a seamless connection between our system and the OBD devices.

5.2 Data Processing

Data obtained from the OBD system will be processed on the backend to identify and present fault information to users through the website and app interfaces.

**6. Database Structure**

Our MongoDB database will store user profiles, fault check results, and other relevant information in a structured manner to ensure data integrity and accessibility.

### **AWS Cloud Services:**

* + Utilize AWS services for scalable and reliable cloud infrastructure.
    - **Amazon EC2 Instances:**
      * Host the application server.
    - **Amazon S3:**
      * Store static assets, such as images or user uploads.
    - **Amazon CloudFront:**
      * Content delivery network (CDN) for fast content delivery to users globally.
    - **AWS Lambda:**
      * Serverless compute for executing code in response to events.

1. **Database: MongoDB (NoSQL):**
   * MongoDB is chosen for its flexibility, scalability, and ability to handle JSON-like documents.
   * **Collections:**
     + **Users:**
       - Store user profiles, including authentication details.
       - Example fields:
         * **user\_id**, **username**, **email**, **password\_hash**, **role**.
     + **FaultChecks:**
       - Store information related to fault checks.
       - Example fields:
         * **check\_id**, **user\_id** (foreign key), **car\_make**, **car\_model**, **timestamp**, **fault\_details**.
2. **Indexes:**
   * Create indexes for efficient data retrieval.
   * Example indexes:
     + **Users** collection: Index on **username** for quick user lookups.
     + **FaultChecks** collection: Compound index on **user\_id** and **timestamp** for efficient fault check history retrieval.
3. **AWS DynamoDB (Optional):**
   * Consider using DynamoDB for specific use cases, especially if you need a fully managed, serverless, and highly scalable NoSQL database.
   * Example tables:
     + **UserSessions:**
       - Store active user sessions for quick authentication checks.
       - Example attributes: **session\_id**, **user\_id**, **expiration\_time**.
4. **Security:**
   * + Implement AWS Identity and Access Management (IAM) for controlling access to AWS services.
     + Enable encryption at rest for data stored in S3 and DynamoDB.
     + Implement SSL for secure communication between the application server and the MongoDB database.
5. **Scalability:**
   * + Utilize AWS Auto Scaling to automatically adjust the number of EC2 instances based on demand.
     + Consider DynamoDB's automatic scaling for handling varying workloads.
6. **Backup and Recovery:**
   * + Implement regular backups using AWS Backup for data stored in S3 and DynamoDB.
     + Snapshot the MongoDB database for backup and recovery.
7. **Monitoring and Logging:**
   * + Use AWS CloudWatch for monitoring and logging.
     + Monitor key metrics such as CPU usage, memory usage, and request latencies.
     + Set up alarms for critical events.

**7. Data Security**

To prioritize user data security, we will implement encryption mechanisms for data transmission and storage. Secure communication protocols will be employed to protect sensitive information.

**8. User Interface (UI) and User Experience (UX)**

The UI/UX design will focus on simplicity and intuitiveness, ensuring a seamless experience for users conducting fault checks and accessing diagnostic results.

**9. Testing and Quality Assurance**

A rigorous testing process, including unit testing, integration testing, and user acceptance testing, will be implemented to guarantee the reliability and accuracy of our system.

**10. Scalability and Performance**

Our system is designed with scalability in mind, utilizing cloud services to handle increased user loads. Performance optimization strategies will be implemented for a smooth user experience.

**11. Deployment and Hosting**

The system will be deployed on cloud infrastructure, and continuous integration tools will automate deployment processes for both the website and mobile app.

**12. Maintenance and Future Development**

We are committed to ongoing maintenance and support. Future development plans include the integration ofadditional diagnostic features and expanding compatibility with a wider range of car models.

**13. Conclusion**

In conclusion, the outlined system structure provides a robust foundation for the development of our car fault check platform. We are enthusiastic about the potential of our startup to simplify car diagnostics and enhance user experiences in the automotive industry.