Automotiq.ai: Advanced Vehicle Diagnostics with Gemma 3n

Executive Summary

Our project is a comprehensive vehicle diagnostic system. It utilizes on-device AI inference to provide drivers with intelligent vehicle health monitoring and troubleshooting. The app leverages Google's Gemma 3n multimodal AI model to analyze diagnostic trouble codes (DTCs) streamed from an OBD2 dongle via Bluetooth Low Energy (BLE) and provides insights on vehicle faults as well as repair guidance.

Technical Architecture

Application Architecture Overview

The application follows a standard Flutter architecture:

1. **Presentation Layer**: Flutter widgets and screens
2. **Business Logic Layer**: Provider-based state management
3. **Data Layer**: Firebase Firestore for cloud storage, SQLite for local DTC database, SharedPreferences for model state caching
4. **Service Layer**: BLE communication, AI model management
5. **Infrastructure Layer**: Authentication, permissions, logging

Provider Architecture and Initialization Sequence

Provider Hierarchy with Dependency Injection

The app implements a provider hierarchy that ensures proper initialization order. The `ModelProvider` starts immediately (non-lazy) to begin model download, while downstream providers wait for their dependencies:

1. **ModelProvider**: Starts model download immediately, highest priority
2. **UnifiedBackgroundService**: Singleton service for background processing
3. **AppAuthProvider**: Waits for `ModelProvider.isModelDownloaded` before triggering anonymous sign-in
4. **UserProvider**: Waits for `AppAuthProvider.user` before initializing user data
5. **VehicleProvider**: Waits for `AppAuthProvider.user` before loading vehicle data

6. **BluetoothManager**: Waits for `AppAuthProvider.user` before initializing BLE services

Initialization Flow and Dependency Chain

The initialization follows a strict dependency chain where each provider waits for its dependencies. The `ModelProvider` begins model download immediately on creation, checking for existing models before downloading. Once the model is ready, `AppAuthProvider` triggers anonymous authentication. After successful authentication, `UserProvider` initializes user data, creating new user profiles for first-time users. Finally, `VehicleProvider` and `BluetoothManager` initialize vehicle and BLE services respectively.

User Routes and Navigation Flow

Root Screen Navigation Logic

The `RootScreen` implements the main navigation logic based on initialization states. It first checks if the model is downloading or not downloaded, showing a splash screen during this process. If model download fails, it displays an error screen with retry functionality. Once the model is ready, it proceeds to authentication. If authentication fails, it displays an error screen. Finally, if the user is authenticated and initialized, it shows the home screen; otherwise, it defaults to the login screen.

Vehicle Setup for New Users

New users start in demo mode. The `HomeScreen` detects when no vehicles are present and shows an empty state with setup guidance. Users can add their first vehicle through the OBD setup flow, which guides them through device pairing and vehicle configuration.

Startup Sequence and State Management

Application Startup Flow

The complete startup sequence begins with environment variable loading and Firebase initialization. The provider hierarchy starts with the `ModelProvider` beginning model download immediately. Other providers wait for their dependencies before initializing, ensuring proper resource management.

Model Download and Initialization

The model download process is managed with progress tracking through the `SplashScreen`. The system checks for existing models before downloading. The model initialization begins in the background immediately after authentication for smoother user experience.

Authentication and User Loading

After model download, authentication proceeds automatically with anonymous sign-in. Users have the option to link an email/password to their account or remain anonymous.

Gemma 3n Integration Architecture

Overview

The Gemma 3n integration was achieved using multiple layers of abstraction that manage AI model lifecycle, concurrent inference operations, and provide seamless integration between chat and diagnostic functionalities. The app uses the `flutter_gemma` dependency to interface with Google's Gemma 3n models while implementing robust queue management to ensure only 1 inference is launched at a time.

Global Agent

The system utilizes a single model for all app actions (Global Agent). This model handles both vehicle diagnostics and user questions, ensuring it has a rich context of the vehicle at all times.

Single Inference Locking Mechanism

The `UnifiedBackgroundService` implements a queue-based approach where all inference requests are queued and processed sequentially. When a request arrives, it's added to the queue and the system checks if the agent is busy. If the agent is available, processing begins immediately. If not, the request waits in the queue. This ensures that only one inference operation runs at any given time, preventing resource conflicts and memory issues.

Core Components

Model Service Layer (model_service.dart)

The `ModelService` class handles the complete lifecycle of Gemma 3n models, from download to inference. It manages model downloads with resumable capability in case downloads are interrupted. The service checks for existing models before downloading, optimizing startup time. Model initialization includes backend selection (GPU/CPU), context window configuration, and multimodal support setup.

Model Provider Layer (model_provider.dart)

The `ModelProvider` acts as the state management layer for the AI model, coordinating between the service layer and UI components. It manages model lifecycle states including downloading, initialization, and chat setup. The provider creates a global chat agent that's shared across the application, handling message processing with streaming responses.

Key Features

1. **Single Inference Locking**: Queue-based system ensures only one inference runs at a time
2. **Streaming Responses**: Real-time token streaming for responsive UI
3. **Multimodal Support**: Image and text processing capabilities
4. **Background Processing**: Non-blocking inference operations
5. **State Persistence**: Chat history and diagnosis results are saved locally
6. **Error Recovery**: Graceful handling of inference failures and queue management
7. **Resource Management**: Proper cleanup and memory management

BLE/OBD2 Component

Overview

The system uses a BLE OBD2 dongle to interact with the vehicle. The BLE/OBD2 component implements a three-layer architecture that provides robust BLE communication with OBD2 devices using the ELM327 protocol.

The architecture follows a provider → service → adapter pattern.

Architecture Layers

1. Provider Layer (BluetoothManager)

The `BluetoothManager` acts as the top-level provider that manages the overall BLE state and coordinates between the UI and underlying services:

The provider layer handles:

1. **State Management**: Connection state tracking and UI notifications
2. **Service Coordination**: Orchestrating BLE operations

2. Service Layer (BleService)

The `BleService` implements the core BLE functionality and manages the low level BLE adapter:

The service layer provides:

1. **Connection Management**: Device discovery, connection, and disconnection
2. **State Streaming**: Real-time connection state updates via RxDart streams
3. **Permission Handling**: BLE permission management

3. Adapter Layer (ReactiveBleAdapter)

The adapter layer uses the `flutter_reactive_ble` plugin to provide platform-specific BLE functionality. An abstract adapter class was created to aid in mocking for unit tests.

ELM327 Protocol Integration

The OBD2 communication service implements the ELM327 protocol for standardized vehicle diagnostics. The ELM327 integration provides:

1. **Standardized Commands**: AT commands for device configuration
2. **PID Requests**: Standard OBD2 Parameter IDs for vehicle data
3. **Response Parsing**: Structured parsing of ELM327 responses

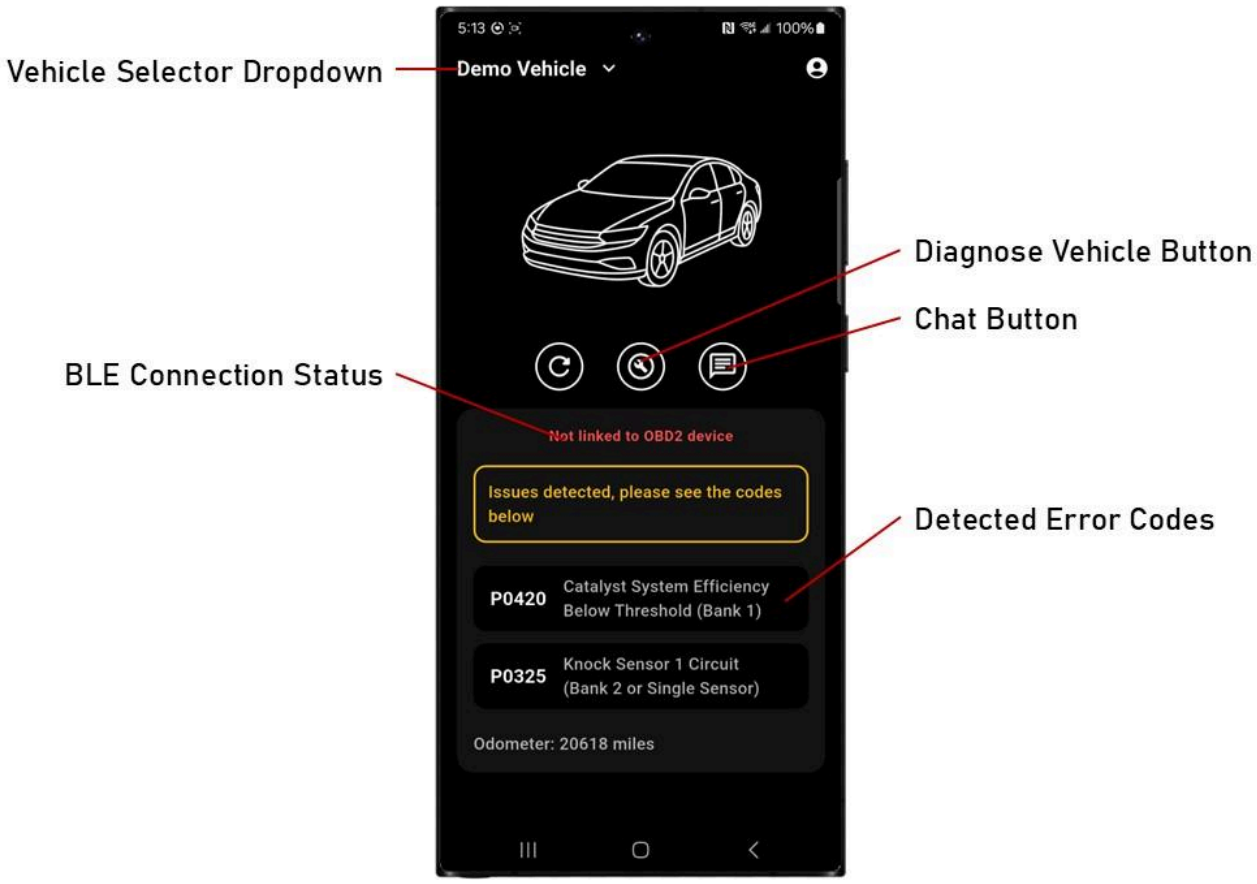
Key Features

1. **Provider-Service-Adapter Pattern**: Clean separation of concerns with proper abstraction layers
2. **ELM327 Protocol**: Standardized OBD2 communication protocol support
3. **Error Recovery**: Error handling and automatic reconnection logic

The BLE component provides a robust foundation for OBD2 device communication while maintaining clean architecture and ensuring reliable operation across different mobile platforms.

User Screens

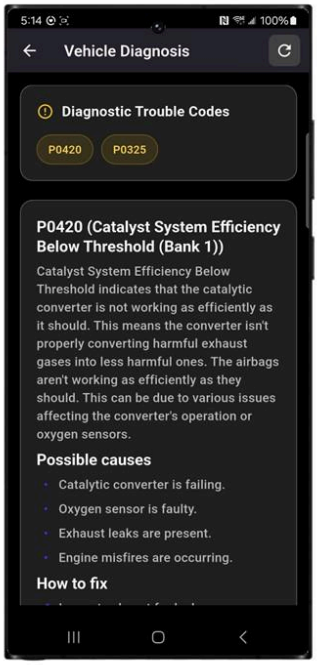
1. Home Screen



The main dashboard displays the selected vehicle with connection status, diagnostic trouble codes, and vehicle information. Features action buttons for data refresh, diagnostics, and AI chat access. Implements a scroll-based interface with vehicle image placeholder and real-time BLE connection status. Automatically attempts device connections when vehicles are selected and shows empty state guidance for new users.

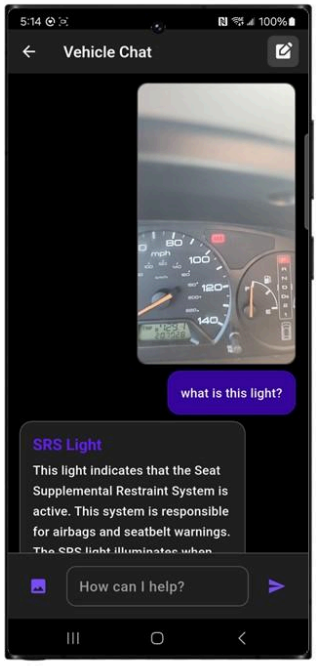
2. Diagnosis Screen

AI-powered analysis interface for diagnostic trouble codes. Displays DTCs with color-coded indicators and generates comprehensive reports using Gemma 3n. Shows real-time processing status with queue information and progress indicators. Users can re-run diagnostics or clear results. Integrates with background service for inference management and provides error handling with retry mechanisms. Results use markdown rendering for rich text display.



3. Chat Screen

Interactive AI assistant for automotive questions and visual diagnostics. Supports text messages and image uploads using Gemma 3n's multimodal capabilities. Displays conversation history with markdown formatting and includes image capture functionality. Integrates with background service for message processing and shows real-time inference status. Provides streaming responses and handles function calls for structured AI responses.



Data Management

1. Local Database (SQLite)

The app includes a SQLite database for diagnostic trouble code information. The database stores DTC codes and quick descriptions. It's populated from a JSON file during app initialization. The database supports efficient querying by DTC codes and includes error handling for corrupted entries. This local storage ensures offline access to diagnostic information and reduces network dependencies for core functionality.

2. Cloud Storage (Firebase Firestore)

Firebase Firestore serves as the primary cloud storage solution for user data and application state. It stores user profiles with authentication information and preferences, vehicle data including device associations and vehicle specifications, and diagnostic history for persistent analysis results.

3. Local Storage (SharedPreferences)

SharedPreferences caches chat message history for conversation continuity. This storage mechanism ensures fast access to frequently used data and provides offline functionality for core app features. The implementation includes proper error handling and data serialization for complex objects.

Technical Challenges and Solutions

1. Concurrent AI Operations

Challenge: Managing multiple simultaneous AI requests without overwhelming the system.

Solution:

1. Queue-based request management
2. Background processing with UI updates
3. Proper state management for inference status
4. Graceful error handling and recovery

2. BLE Connection Reliability

Challenge: Maintaining stable BLE connections with OBD2 devices.

Solution:

1. Connection state monitoring

2. Automatic reconnection logic
3. Error recovery mechanisms

3. Model Download and Management

Challenge: Large AI models (3-6GB) need efficient download and storage management.

Solution:

1. Model download upon app startup (first time only)
2. Local file storage with existence checking

4. Offline Persistence

Challenge: Handling a fully offline app, while keeping clean flows for authentication and multi-device usage requires careful offline handling.

Solution:

1. Firebase built-in offline persistence
2. Authenticate once while online, then never again
3. Gemma 3n model is designed to be offline

Future Enhancements

1. Cross Platform Support

Our current app supports Android. Since we have built this app using Flutter, we expect to be able to provide cross-platform support to iOS devices as well.

2. Predictive maintenance

Given more extensive sensor data regarding vehicle health, LLMs should be able to predict when certain faults will occur and recommend pre-emptive action. We hope to integrate further sensors with vehicles so that we may leverage the full potential of Gemma for fault prediction.

3. On the fly Diagnosis

Our app currently diagnoses faults based on DTCs provided by the vehicle when the user enters the diagnosis screen. In future, we hope to poll the vehicle periodically, perform inference using Gemma in the background and notify users immediately when a fault occurs.

4. Developing for Business Customers

The central goal of this project was to create an effective diagnosis tool for drivers. While this need was met, small businesses (fleet owners) require additional features and management software. The next step of this project is to develop the product for business use.

Conclusion

Automotiq represents a significant advancement in automotive diagnostics by combining cutting-edge AI technology with practical vehicle monitoring capabilities. The integration of Gemma 3n provides users with intelligent, contextual assistance to offer actionable insights and guidance.

The app's success lies in its ability to bridge the gap between complex automotive diagnostics and user-friendly interfaces, making vehicle health monitoring accessible to everyone while providing the depth and accuracy required for serious automotive work.