1. Executive Summary & Project Intent

1.1. Introduction to Libre Automotive Diagnostics

The **Portable Libre Diagnostic** project is an open-source initiative designed to provide a highly reliable, accessible, and transparent automotive diagnostic framework. Historically, vehicle diagnostics have been gated behind expensive, closed-source proprietary software and hardware tools. This restriction limits vehicle owners, independent mechanics, and research engineers from fully auditing and interacting with their own vehicle data.

By delivering a native Android application built entirely on standard, verifiable software patterns and a pure Java architecture, this project establishes a modular client capable of directly interrogating vehicle networks without hidden commercial middleware. The application leverages standard commercial-off-the-shelf (COTS) ELM327 Bluetooth hardware adapters to bridge the gap between portable mobile computing and internal vehicle electronics.

1.2. NLnet NGI0 Commons Fund Compliance Statement

This software client is developed under the strict open-source and digital commons mandates of the **NLnet NGI0 Commons Fund**. The primary objective under this funding compliance framework is the delivery of software that is:

- **Fully transparent:** No obfuscated dependencies or closed source binaries.
- **Privacy-respecting:** Zero telemetry tracking, cloud-phoning, or user monetization.
- **Architecturally modular:** Designed as a digital commons asset so that external developers can fork, extend, or adapt the codebase for non-standard or highly specialized hardware environments without licensing friction.

All deliverables, including repository build baselines, architectural documentation, and communication protocol parsers, are formatted to fulfill the technical milestones required for public auditability and open integration.

1.3. Target Vehicle Integration Capabilities (SAE J1979 / ISO 15765-4)

The application acts as a hardware-agnostic interpreter targeting standard legislated vehicle registers. The core system capabilities focus on:

- **SAE J1979 (OBD-II Standard):** Implementation of universal diagnostic services (Service 01 for real-time telemetry streaming, Service 03 for Diagnostic Trouble Code retrieval, and Service 04 for emission-related fault clearance).
- **ISO 15765-4 (CAN Bus Protocol):** Architectural pre-arrangements to support raw data packet transmission over high-speed Controller Area Network (CAN) physical links (500 kbps and 250 kbps carrier frequencies), utilizing standard 11-bit identifiers as well as foundational compatibility wrappers for 29-bit extended frame addressing.

---

2. System Architecture & High-Level Design

2.1. Architectural Pattern Selection (Clean Architecture & MVVM)

The application utilizes **Clean Architecture** combined with the **MVVM (Model-View-ViewModel)** design pattern. This approach isolates volatile vehicle hardware operations from the Android OS layout engines. By grounding core operations in pure Java, the business rules remain decoupled from third-party framework updates. This guarantees long-term maintainability as a digital commons asset.

2.2. Boundary Segregation (Presentation, Domain, Data Layers)

The codebase enforces rigid software boundaries with unidirectional, inward-pointing compilation dependencies:

- **Presentation Layer:** Contains UI controllers (`Activities`) and state managers (`ViewModels`). It tracks lifecycle events and renders data but contains zero knowledge of raw serial streams or OBD-II hex calculations.
- **Domain Layer:** The framework-agnostic core containing business models (`VehicleMetric`) and interface abstractions (`ObdCommand`). It does not import the Android SDK.
- **Data Layer:** The network and hardware infrastructure containing the Hardware Abstraction Layer (`BluetoothManager`) and low-level byte utilities (`ResponseParser`).

2.3. Package Topology and Directory Tree Structure

The project configuration separates layer boundaries directly within the filesystem under `app/src/main/java/com/librediagnostic.portable/`:

```
📁 core/
 ├── 📁 bluetooth/
 │    ├── 📄 BluetoothDeviceModel.java  # Wraps discovered hardware specifications
 │    ├── 📄 BluetoothManager.java      # Manages RFCOMM socket lifecycles & channels
 │    └── 📄 DiagnosticState.java       # Thread-safe global session state machine
 └── 📁 parser/
      ├── 📄 ObdCommand.java            # Structural contract for automotive messages
```

```
      ├── 📄 EngineRpmCommand.java      # Concrete SAE J1979 Service 01 PID 0C logic
      └── 📄 ResponseParser.java        # Telemetry stream multiplexer & error validator
▫ presentation/
 └── 🗁 dashboard/
      ├── 📄 DashboardActivity.java     # Android View handling layout rendering
      └── 📄 DashboardViewModel.java    # StateHolder exposing reactive LiveData streams
```

2.4. Cross-Layer Dependency Constraints and Abstraction Barriers

To eliminate architectural erosion, three structural validation limits are established:

1. The presentation domain interacts with lower layers solely through lifecycle-safe, reactive **`LiveData` state observers**.
2. No class inside the `core` or `domain` spaces is allowed to import structural Android framework dependencies (`android.view`, , or context contexts).
3. Component communication remains decoupled via interface patterns. The view models dispatch execution payloads using abstract interfaces, blocking upper-level components from seeing raw hex streams or physical layer transactions.

---

3. Communication Scheduling & Hardware Data Flow

3.1. Asynchronous Multithreading Model (Executor Services)

Automotive data acquisition requires low, predictable latency loops, making execution on the Android Main/UI thread strictly unacceptable. The architecture utilizes a managed **`ExecutorService`** configured with a dedicated thread pool layout. This mechanism isolates raw hardware blockages from visual rendering systems, establishing thread-safety during prolonged streaming sessions.

3.2. Serial RFCOMM Socket Initialization & Resource Lifecycles

Physical data links rely on a classic **Bluetooth RFCOMM serial socket connection** bound to the standard Serial Port Profile (SPP) UUID: `00001101-0000-1000-8000-00805F9B34FB`. The hardware link manager controls resource state transitions through a deterministic sequence:

1. **Discovery Interruption:** Active hardware scans are forced to stop via `cancelDiscovery()` prior to connection to preserve maximum radio transceiving bandwidth.
2. **Channel Allocation:** Low-level standard input and output byte streams (`InputStream` and `OutputStream`) are bound to the opened socket.
3. **Graceful Teardown:** Upon session closing or interface connection degradation, streams are closed sequentially and nullified within protected `finally` contexts to avoid hardware file-descriptor leakage.

3.3. Unidirectional Data Flow (UDF) Loop Mechanics

To secure state predictability, the telemetry transmission framework operates via an isolated, enclosed **Unidirectional Data Flow (UDF)** loop:

```
[DashboardViewModel] ──(Dispatches Loop Request)──> [BluetoothManager]
         ▲                                                  │
         │ (Posts State Mutation via LiveData)        (Transmits ASCII Command)
         │                                                  ▼
[DashboardActivity] <──(Triggers View Update)── [ResponseParser] <── [ELM327 / ECU]
```

- The presentation component dispatches a scheduled execution interval event.
- The transaction travels outward through the Hardware Abstraction Layer, interfaces with the bus link, and returns as a raw hexadecimal stream response.
- The payload goes directly to the processing matrix, updating the reactive `LiveData` wrapper to notify UI view elements on the main thread safely.

3.4. Byte-Stream Buffering and Telemetry Packet Distribution

Raw diagnostic metrics arrive asynchronously inside the `InputStream` byte buffer. Because ELM327 chipsets terminate message lines using carriage return and prompt character tokens (`\r` or `>`), the software layer continuously buffers inbound chunks until these specific structural markers are intercepted. Once a coherent sentence is captured, it is packaged as a clean alphanumeric string payload, stripped of system-level transmission noise, and dispatched to the decoding framework.

---

4. Vehicle Protocol Translation & Decoupling

4.1. ELM327 Command Sequence Pipeline (AT Instructions)

Before querying real-time telemetry from the vehicle's Electronic Control Units (ECUs), the application initiates a sequential handshake with the ELM327 integrated circuit. This initialization pipeline utilizes specialized **Hayes AT commands** to configure the serial-to-CAN conversion parameters.

The standard initialization stack executes the following structural sequence:

1. **`AT Z` (Reset):** Restores the chip to its default state, clears internal buffers, and resets baud rates.
2. **`AT SP 0` (Automatic Protocol Detection):** Instructs the ELM327 to automatically discover the underlying physical layer protocol (e.g., , , or).
3. **`AT H1` (Headers On):** Forces the chip to display protocol headers (CAN IDs) in responses, which is necessary for parsing multi-ECU communications and extended frame identifiers.
4. **`AT E0` (Echo Off):** Disables character mirroring to reduce bandwidth consumption on the serial line.

4.2. On-Board Diagnostics (OBD-II) PID Multi-Register Mapping

The application routes diagnostic requests using structural **Parameter IDs (PIDs)** mapped across standard diagnostic modes defined by the SAE J1979 specification. Every standard request frame contains a combination of the Diagnostic Service Mode and the specific hex ID assigned to the desired automotive sensor.

Diagnostic Mode (Hex)	Parameter ID (PID - Hex)	Target Metric Profile
01 (Live Telemetry)	0C	Engine RPM
01 (Live Telemetry)	0D	Vehicle Speed
01 (Live Telemetry)	05	Coolant Temperature
03 (Fault Extraction)	N/A (Full Register Dump)	Active Trouble Codes
04 (Fault Clearance)	N/A (Clear Frame Request)	Clear MIL & Error Codes

4.3. Standard Bitstream Hexadecimal Decoding Formulations

When a positive acknowledgement is received from the vehicle bus, the raw hexadecimal string is mapped against the standardized formulas mandated by the SAE framework. A positive response mirrors the service mode by adding `0x40` to the request byte (e.g., a Mode `01` query returns a packet starting with `41`).

The application incorporates strict mathematical parsing boundaries to process these byte streams:

- **Engine RPM (PID 0C):** Decodes a 2-byte response via the formula: `((Byte A * 256) + Byte B) / 4`. This resolves a resolution matrix ranging from `0` to `16,383.75 rpm`.
- **Vehicle Speed (PID 0D):** Decodes a single-byte payload using the metric notation: `Byte A` directly, providing an integer scale from `0` to `255 km/h`.
- **Engine Coolant Temperature (PID 05):** Evaluates a 1-byte temperature offset signature: `Byte A - 40`, mapping thermal performance from `-40°C` to `+215°C`.

4.4. Brand-Specific Extended CAN Identifier Adaptability

While universal OBD-II commands use standard 11-bit CAN identifiers (typically `0x7DF` for functional broadcast requests), modern vehicle diagnostics often require deeper inspection through non-standard manufacturer-defined registers.

To bridge this gap, the software architecture features a decoupled data abstraction layer that can adapt to **29-bit extended CAN identifier frames**. By issuing initialization overrides such as `AT CP XX` (CAN Programmed Flow Control) and configuring custom message headers via `AT SH XXX` (Set Header), the core engine can dynamically modify its transmission identifiers. This structural separation allows external brand modules to easily switch communication channels between standard powertrain buses and specific proprietary control loops, such as body control (BCM) or safety modules (SRS), without altering the underlying core serial processing architecture.

---

5. Defensive Engineering & System Fault Tolerance

5.1. Hardware Link Degradation & Drop Handling (IOException Execution)

The physical environment of a moving vehicle is highly prone to unexpected hardware disconnection due to vibrations, electrical interference, or distance dropouts. The application addresses this volatility via structural exception guarding:

- **Active Link Polling:** The background streaming threads constantly evaluate the readability of the `InputStream`.
- **Graceful Degradation:** If the serial link drops, a low-level `IOException` is caught by the execution layer. The engine triggers an immediate transition of the global state to `DiagnosticState.ERROR`.
- **Safe Teardown Pipeline:** A strict `finally` block instantly closes and nullifies the `BluetoothSocket` descriptors, resetting memory allocations to prevent file-descriptor exhaustion and ensuring the app can immediately enter a safe reconnection back-off loop.

5.2. Vehicle Bus Congestion and ECU Exception Containment Matrix

During high-load states (such as active engine cranking or network traffic spikes on the CAN Bus), the vehicle's ECUs might temporarily drop diagnostic requests or return hardware errors. Rather than allowing these conditions to cause application crashes, the parsing framework treats them as valid string signals:

ECU Error String	Detected Failure Mode	Software Containment Action
"NO DATA"	Sensor missing or inactive	Returns "0", marks field
"CAN ERROR" / "BUS ERR"	Physical bus noise/fault	Clears buffer, logs drop
"STOPPED"	Session timed out by ECU	Re-runs basic initialization
"BUS INIT ERROR"	Protocol synchronization	Resets OBD protocol stack

The system uses a containment pattern where all incoming strings pass through `ResponseParser.isErrorFrame()`. If a fault string is flagged, the app returns a safe fallback value (`0`) to prevent UI components from parsing empty fields, keeping the data loop alive while waiting for the next clean frame.

5.3. Operational System Fail-Safe States (Diagnostic Lifecycle Enum)

To enforce strict determinism, the entire client runs on a centralized **Finite State Machine (FSM)** driven by `DiagnosticState`. The presentation layer is strictly decoupled from the internal communication state; it simply observes this state machine. If an unexpected error drops the system state into `DiagnosticState.ERROR`, the UI automatically shifts into a fail-safe configuration—freezing live fields, displaying an alert overlay, and blocking user interaction with diagnostic clear functions (`Service 04`) until the core manager signals a safe, verified transition back to `CONNECTED_READY`.

5.4. OS-Level Sandbox Security and Android Permission Interception

Operating within the Android sandbox requires robust, proactive permission management to prevent runtime security exceptions (`SecurityException`). The application enforces clean architectural security gates:

- **Pre-Flight Verification:** Before invoking the `BluetoothManager`, the client checks for the necessary runtime flags (`BLUETOOTH_CONNECT`, `BLUETOOTH_SCAN` on Android 12+ / API 31+, and `ACCESS_FINE_LOCATION` for older targets).

- **Graceful Interception:** If permissions are revoked mid-session, the application intercepts the resulting system error, halts the background `ExecutorService` tasks immediately, and shifts the UI into a clean, explanatory guidance layout. This structure ensures that hardware access routines can never run without proper system clearance, keeping the application entirely safe from unexpected OS-enforced termination.
