📄 Phase 7.4 – Edge vs Off-Board Trade-Offs (UNO Q as the Edge Node)

7.4.1 Context and Goal

By this point the **Arduino UNO Q**–based system can:

- Listen to the vehicle CAN bus
- Decode voltages from three "virtual ECUs" via `harness_demo.dbc`
- Apply **rule-based harness fault logic** (Harness A/B/C) in real time
- Run **simple statistical anomaly detection** (drift, trend, early warnings)
- Do all of this **live**, on the edge, with no cloud dependency

Phase 7.4 answers a key architectural question:

> **What should stay on the edge (UNO Q), and what could/should be done off-board (backend, cloud, or engineering laptop)?**

We're designing this like a **small telematics / condition monitoring module** that you could deploy into a vehicle harness to monitor degradation over its lifetime.

7.4.2 What "Edge" Means in This Project

In this project, *edge* means:

- Hardware: **Arduino UNO Q**
- Environment: On-vehicle, connected to the CAN bus
- Constraints:
    - Limited CPU and RAM compared to a server

    - Limited local storage
    - Potentially intermittent connectivity
    - Needs to be **robust**, **predictable**, and **explainable**

The edge device:

- Sees **all raw CAN traffic** (or at least selected frames)
- Decodes signals
- Takes **fast, local decisions** about anomalies and alerts

7.4.3 Functions That Must Stay On the UNO Q (Edge)

These are the responsibilities that **must** remain on the UNO Q for this project to make sense.

1. CAN acquisition and DBC decoding

- Reading the CAN frames via the MCP2515
- Decoding `ECUA_Supply_Voltage`, `ECUB_Supply_Voltage`, `DCDC_Output_Voltage` using `harness_demo.dbc`
- Validating message presence and timing (is the ECU still alive?)

This is **fundamental** edge logic. If decoding depends on the cloud, the system becomes fragile and laggy.

2. Core rule-based harness diagnostics

The **hard rules** that identify:

- Harness A fault pattern
- Harness B fault pattern

- Harness C fault pattern

These rules are:

- **Deterministic**
- Easy to explain to a safety engineer or technician
- Low CPU and memory cost
- Essential for immediate, local safety/maintenance decisions

So the UNO Q must keep:

- The delta calculations (ΔA, ΔB)
- The voltage band checks (e.g. DCDC ≈ 14 V, ECU below by > X V)
- The logic that raises `[ALERT][HARNESS_X] ...` messages

3. Lightweight statistical anomaly detection

The statistical layer we built in Phase 7.3 is intentionally:

- Simple (means, EWMA, approximate trends)
- Bounded in RAM (sliding window of recent samples)
- Deterministic and transparent

Those features are **small and cheap enough to live on the UNO Q**, and they offer:

- Early drift detection
- Noise/volatility detection
- Simple "time to threshold" style estimations (RUL-like)

Keeping this on the edge means:

- The system can warn about deterioration **even if it never phones home**
- You can diagnose issues from the **vehicle logs alone** (App Lab console / serial / local log files)

7.4.4 What Makes Sense Off-Board

Some functionality is **better off-board** (engineering laptop, cloud, or backend service):

1. Heavy analytics and model development

Things that are *not* worth deploying onto the UNO Q:

- Training advanced anomaly detection models (e.g. machine learning)
- Calibrating complex degradation models
- Running big statistical analyses over **fleet data**

Instead, the workflow can be:

1. **Edge** logs derived metrics (e.g. ΔA, ΔB, EWMA, trends, alert events)
2. Logs are uploaded during service or via telematics
3. **Off-board tools** crunch months of data from many vehicles
4. Outcome:
    - Updated thresholds
    - Improved rules
    - New statistical patterns—then pushed back to the firmware as **simplified logic**

2. Long-term storage

The UNO Q is not a long-term data warehouse.

Off-board responsibilities:

- Keeping **months/years** of CAN/diagnostic history
- Supporting advanced queries:
  - “Show me all vehicles with increasing ΔA over the last month”
  - “Compare this vehicle to the fleet median”

Local device:

- Stores only **short windows** of raw data for debugging
- Or a rolling log with size limits (overwriting old data)

3. Visualization and interactive diagnostics

Real-time plotting, dashboards, and advanced UI belong:

- On a laptop
- In App Lab
- In a browser dashboard or cloud tool

The UNO Q’s role is to emit structured events:

- `[STATS] ...`
- `[ALERT][HARNESS_A] ...`
- `[EARLY][HARNESS_A_DRIFT] ...`

…and the off-board tools turn this into:

- Time series plots
- Comparative fleet views
- Operator dashboards

7.4.5 Data That Should Be Logged and/or Uplinked

To bridge edge and off-board analysis, we define the **minimum useful data** to log/uplink:

1. **Key signals and deltas**
   - `V_A`, `V_B`, `V_DCDC`
   - ΔA, ΔB
   - Possibly EWMA values and trending indicators
2. **Alert events**
   - Timestamp
   - Harness ID (A/B/C)
   - Snapshot of voltages and deltas at the time
3. **Environment / meta (future extension)**
   - Temperature, load conditions (if available)
   - Software version, device ID, mileage if accessible

Logged locally as **CSV** (as we prototyped in Phase 6), and optionally sent to:

- Engineering laptop
- Cloud backend
- On-premises diagnostic system

7.4.6 Security, Updates, and Safety Considerations

Because the UNO Q is acting like a **telematics / monitoring module**, we need to be clear about:

- **Separation of concerns**
  - The device is **monitoring**, not controlling safety-critical actuators
  - It listens on CAN and raises alerts, but does not directly control ECUs
- **Update strategy**
  - Edge rules and thresholds may be updated as we learn from data
  - Updates should be versioned and traceable (which rules were active when a certain alert was raised?)
- **Fail-safe behaviour**
  - If decoding fails (missing DBC, corrupt CAN data), system should:
    - Log the failure
    - Avoid making incorrect diagnoses
    - Possibly fall back to simpler “heartbeat” checks

7.4.7 Summary and Bridge to Phase 8

By the end of Phase 7.4, we have a clear split:

- **On the UNO Q (edge):**
  - CAN acquisition & DBC decoding
  - Rule-based harness diagnostics
  - Lightweight statistical drift analysis
  - Local alert generation and logging

- **Off-board:**
  - Long-term storage and fleet analysis
  - Model refinement and advanced analytics
  - Rich visualization and dashboards

This sets us up for **Phase 8 – Event Handling & Alerts**, where we treat alerts as **first-class events** with states and transitions (raised, acknowledged, cleared), and later explore how to notify a user (e.g. via logs, UI, or in a production system, email / integration).