PDF 5.1 – Python Application Architecture

1. Purpose

This document defines the **high-level architecture of the Linux/Python application** running on the Arduino UNO Q's microprocessor.

The Python app is responsible for:

1. Receiving CAN frames forwarded by the MCU via the Arduino Bridge
2. Decoding frames into signals using a DBC
3. Evaluating predictive maintenance rules (harness resistance scenario)
4. Logging data for offline analysis and replay
5. Emitting events/alerts when conditions are met

Implementation details and full source will live under:

- `python/app/`
- `python/rules/`
- `python/logging/`
- `python/dbc/`

This document focuses on **structure, responsibilities, and data flow**, not line-by-line code.

---

2. Overview

At a high level, the Python app is organised into five main components:

1. **BridgeReceiver** – receives `can_frame_v0` messages from the MCU
2. **FrameRouter** – normalises and forwards frames to downstream consumers
3. **Decoder** – converts raw CAN frames into physical signals using the DBC
4. **RuleEngine** – applies predictive maintenance rules on decoded signals
5. **LogManager** – records raw and/or decoded data for offline use

Supporting pieces:

- **AppLoop** – main loop + lifecycle management (via `App.run()`)
- **EventBus** – simple in-process pub/sub for internal events
- **Config** – central configuration (DBC path, logging settings, thresholds)

---

3. Data Model

3.1 Raw CAN Frame (from MCU)

The MCU sends a compact, MsgPack-encoded **v0** structure:

```
can_frame_v0:
  version: uint8 (always 0)
  t_ms:    uint32  (MCU timestamp, ms since boot)
  id:      uint32  (11 or 29-bit CAN ID)
  dlc:     uint8
  flags:   uint8   (bit 0 = extended frame flag)
  data:    8 x uint8
```

On Python side this is represented as a small `CanFrame` object:

```
CanFrame:
  ts_mcu_ms: int
  can_id:    int
  is_extended: bool
  dlc:       int
  data:      bytes or List[int]
```

3.2 Decoded Signal Sample

After DBC decoding, each signal sample is represented as:

```
SignalSample:
  name:       str          # e.g. "ECUA_Supply_Voltage"
  ts_mcu_ms:  int
  value:      float        # physical value (e.g. 13.7)
  unit:       str          # e.g. "V"
  source_id:  int          # CAN ID
```

3.3 PM Rule Event

When a predictive maintenance condition is met, the RuleEngine emits:

```
PmEvent:
  ts_mcu_ms: int
  event_type: str       # e.g. "HARNESS_A_DEGRADING"
  severity:  str        # "info" | "warning" | "critical"
  details:   dict       # context (voltages, deltas, estimated TTT, etc.)
```

These events will later drive notifications (Phase 8).

---

4. Component Responsibilities

4.1 BridgeReceiver

Responsibility:
Terminate the MCU→Linux bridge and convert messages into `CanFrame` objects.

Key tasks:

- Register a handler with `Bridge.provide("can_frame_v0", handler)`
- Map incoming MsgPack arguments → `CanFrame`
- Push frames into an in-memory queue for downstream components

Constraints:

- Must be **non-blocking** and light; it runs in the same process as the App loop
- No heavy decoding or file I/O in the callback

4.2 FrameRouter

Responsibility:
Central “fan-out” stage between the bridge and the rest of the app.

Key tasks:

- Maintain a thread-safe queue `frame_queue`
- Provide non-blocking `get_next_frame(timeout)` to consumers
- Optionally maintain simple statistics (fps, last ID seen)

Why separate from BridgeReceiver?

- Keeps bridge callbacks minimal
- Allows future additions (e.g. filter subsets of frames) without touching bridge glue code

---

4.3 Decoder

Responsibility:
Translate `CanFrame` → one or more `SignalSample` using the DBC.

Internally, it will use:

- A DBC handling library (e.g. `cantools`) or
- A minimal custom decoder for learning purposes

Key tasks:

- Load DBC at startup from `python/dbc/harness_demo.dbc`
- For each frame:
    - Look up message definition by CAN ID
    - Extract and scale signal(s)
    - Emit `SignalSample` objects for each defined signal

For this project's demo:

- `ECU_A_Status` → `ECUA_Supply_Voltage`
- `ECU_B_Status` → `ECUB_Supply_Voltage`
- `DCDC_Status` → `DCDC_Output_Voltage`

Decoded signals are pushed into a **SignalState** structure used by the RuleEngine.

4.4 SignalState

Responsibility:
Keep the latest value and short history for each signal of interest.

Key data:

- `latest_value[name]` → last `SignalSample`
- `history[name]` → ring buffer of recent samples (for slope calculation)

This enables:

- Fast access for rule evaluation
- Trend calculation (dV/dt) over a configurable time window

4.5 RuleEngine

Responsibility:
Implement the **harness resistance predictive maintenance logic**.

Inputs:

- `SignalState` (latest `V_A`, `V_B`, `V_D`)
- Configurable parameters:
  - `ΔV_threshold` (e.g. 0.5 V)
  - `min_persistence_s` (e.g. 5 s)
  - `warning_ttt_s` (time-to-threshold alert window)

Outputs:

- `PmEvent` objects, published to the internal EventBus
- Optional: info-level “health state” (OK / A-degrading / B-degrading / C-degrading)

Core checks (executed in the App loop at fixed rate):

- **Harness A suspect** if:
  - `V_D ≈ nominal`, `V_B ≈ V_D`, `V_A < V_D - ΔV_threshold`
- **Harness B suspect** if:
  - `V_D ≈ nominal`, `V_A ≈ V_D`, `V_B < V_D - ΔV_threshold`
- **Harness C suspect** if:
  - `V_D ≈ nominal`, `V_A ≈ V_B < V_D - ΔV_threshold`

For each suspected harness, compute a simple voltage degradation rate (`dV/dt`) and an estimated **time to reach 9.0 V**.
If `TTT < warning_ttt_s`, raise a **predictive event**.

4.6 LogManager

Responsibility:
Central logging of both raw and processed data.

Log streams:

1. **Raw CAN log**
   - `timestamp, id, dlc, data…`
   - Target format to be refined in Phase 6 (MF4 or similar)
2. **Decoded signal log**
   - `timestamp, signal_name, value, unit`
3. **Event log**
   - `timestamp, event_type, severity, details`

Design choices:

- Logging performed **asynchronously** when possible
- Fail-safe: if logging I/O fails, it must not crash the app
- Log rotation / retention to be defined in Phase 6 (rolling buffer, event pinning)

4.7 EventBus

Responsibility:
A very simple internal pub/sub mechanism.

Example topics:

- `pm_event` – predictive maintenance events
- `logging_error` – non-fatal logging failures

Initially this can be implemented as:

- A list of callbacks per topic
- Synchronous dispatch in the App loop (simple and predictable)

4.8 AppLoop (Main)

Responsibility:
Glue everything together and keep the application alive.

Main loop responsibilities:

- Poll FrameRouter for new frames
- Feed frames into Decoder
- Update SignalState
- Periodically call RuleEngine
- Periodically call LogManager flush/rotation
- Print periodic stats for debugging

It is registered with `App.run(user_loop=main_loop)` from the Arduino RouterBridge Python API.

---

5. Execution Model

5.1 Single-threaded core

To keep things understandable and robust:

- The core of the system (FrameRouter, Decoder, RuleEngine, LogManager) runs in a **single-threaded loop**.
- Bridge callbacks are used only to enqueue data into frame queues.

Benefits:

- Easier reasoning about state (no complicated locking)
- Deterministic behaviour for logs and rules
- Simpler for readers of the project to understand

If needed later, logging can be offloaded to a worker thread, but Phase 5 will stay single-threaded.

5.2 Timing

- **Frame processing:** as frames arrive (driven by MCU → Bridge)
- **Rule evaluation:** fixed cadence, e.g. every 100–200 ms
- **Stats:** once per second (`recv_count`, rules triggered, log size)

This is more than sufficient for voltage signals changing over seconds/minutes.

---

6. Configuration & Extensibility

A simple configuration file `config.yaml` (or Python module) will define:

- DBC path
- Signal names for the demo (`ECUA_Supply_Voltage`, etc.)
- Rule thresholds and time windows
- Logging paths and retention parameters

The architecture is intentionally modular so that:

- New rules can be added without changing the bridge or decoder
- Other demos (e.g. temperature-based PM) can reuse the same structure
- Multiple CAN IDs/signals can be added with minimal changes

---

7. Relationship to Project Phases

- **Phase 4** validated the transport (MCU → Bridge → Python) using synthetic CAN frames.
- **Phase 5** defines the structure that will:
    - receive real frames
    - decode them
    - evaluate rules
    - log results
- **Phase 6** will focus on:
    - concrete log formats (MF4 or equivalent)
    - replay tools
    - retention policy implementation
- **Phase 7–8** will build on this architecture to:
    - refine rules and trends
    - send alerts (e.g. email)
    - demonstrate end-to-end PM behaviour.