📄 P6.3 – Data Replay & Verification

1. Purpose of this Phase

Phase 6.3 demonstrates that:

1. Logged CAN data can be replayed offline.
2. Replay produces **identical decoded results** to live data.
3. Replay enables **verification**, debugging, and analysis without hardware.
4. The storage format is compatible with the predictive maintenance logic introduced in Phase 7.

Replay + verification closes the loop for:

Live CAN → CSV log → Replay → DBC decode → Predictive rules

This confirms that no semantic information is lost between live operation and stored datasets.

---

2. Inputs

This phase consumes:

- **CSV CAN logs** produced in Phase 6.2
    - Format includes timestamps, CAN ID, DLC, data bytes, flags.
    - Hex representation for IDs and bytes for readability.

Example log snippet:

```
2026-01-09T11:41:40.429Z,0x111,8,0x8D,0x00,0x00,0x00,0x00,0x00,0x00,0x00,STD
2026-01-09T11:41:40.744Z,0x112,8,0x8D,0x00,0x00,0x00,0x00,0x00,0x00,0x00,STD
2026-01-09T11:41:40.909Z,0x113,8,0x8D,0x00,0x00,0x00,0x00,0x00,0x00,0x00,STD
```

- **DBC file**
    - `harness_demo.dbc`
- **Replay Utility**
    - `replay_decode.py`

---

3. Replay Mechanism

The replay tool performs:

1. Parse CSV rows
2. Convert stored data → in-memory CAN frames
3. Run through `cantools` decoder
4. Print decoded physical signals + timestamps
5. Optionally stream at real or accelerated time

Implementation intentionally mirrors live decoding, so replay is **functionally identical** but **deterministic**.

---

4. Validation Results

Replay output matched live results:

```
[REPLAY][DECODE] t=2026-01-09 11:41:40.429000 id=0x111 ECU_A_STATUS
ECUA_Supply_Voltage = 14.1 V
[REPLAY][DECODE] t=2026-01-09 11:41:40.744000 id=0x112 ECU_B_STATUS
ECUB_Supply_Voltage = 14.1 V
[REPLAY][DECODE] t=2026-01-09 11:41:40.909000 id=0x113 DCDC_STATUS
DCDC_Output_Voltage = 14.1 V

[REPLAY] Done. total_rows=76, decoded_rows=76
```

Key observations

- 100% replay fidelity
- 0 dropped frames
- Decoded values matched live output
- DBC interpretation identical
- Replay performance > real-time (fast verification)

---

5. Offline Predictive Replay (Cross-Phase Link)

Replay also supports running **predictive maintenance logic offline**.

This establishes:

- Historical reprocessing
- Debugging without hardware
- Scenario simulation

- Regression testing

This capability is important for:

- **Phase 7.3 – Statistical anomaly detection**
- **Phase 7.4 – Edge vs off-board division**
- Field investigations after deployment

---

6. Why Replay is Important

Replay creates three strategic benefits:

A. Engineering Validation

Engineers can verify:

- Correct DBC interpretation
- Signal semantics
- Threshold logic

without injecting hardware faults.

B. Predictive R&D

Offline logs enable:

- Fault modelling
- “What if” simulations
- Voltage degradation studies

C. Field Telemetry Post-Analysis

A deployed telematics unit would:

Store → Upload → Replay + Diagnose

This model mirrors industrial **fleet predictive maintenance** workflows.

---

7. Relationship to MF4 (Phase 6.1–6.2)

Replay reinforces that **our storage format preserves full semantic value** even though we temporarily parked MF4 due to `gcc` constraints in the App Lab Python environment.

The architecture remains MF4-ready:

```
[Live CAN]
     ↓
[DBC Decode + Predict]
     ↓
[Log (CSV now, MF4 future)]
     ↓
[Replay]
     ↓
[Predict + Analysis]
```

Offline replay is therefore future-compatible with `asammdf` once enabled on a different execution environment.

---

8. Conclusion

Phase 6.3 closes Phase 6 successfully. We now have a validated pipeline for:

✔ Live ingest from CAN
✔ DBC decoding
✔ Predictive maintenance logic
✔ Logging
✔ Replay & verification

This foundation is strong enough to advance to **Phase 7**.