Replay, Reason, Refly with Apache Beam

By Charles Adetiloye & Ian MacDonald

BEAM SUMMIT NYC 2026

About the Presenters

Charles Adetiloye

Cofounder & Lead AI Engineer, MavenCode

Charles has over 20 years of experience building large-scale distributed applications, specializing in production-grade GenAI, Agentic AI, and ML platforms.

Ian MacDonald

Full Stack ML/Platform Engineer, MavenCode

Ian specializes in designing and managing production-grade machine learning and LLM-based applications, with a dual role at TAMU-CC ARI.

About MavenCode

MavenCode is an Applied AI Engineering Company based in Dallas. We specialize in training, product development, and consulting services.

AI & ML Infrastructure

Provisioning scalable infrastructure both On-Premise and in the Cloud.

Operationalization

Production-grade development and deployment of ML platforms across hybrid environments.

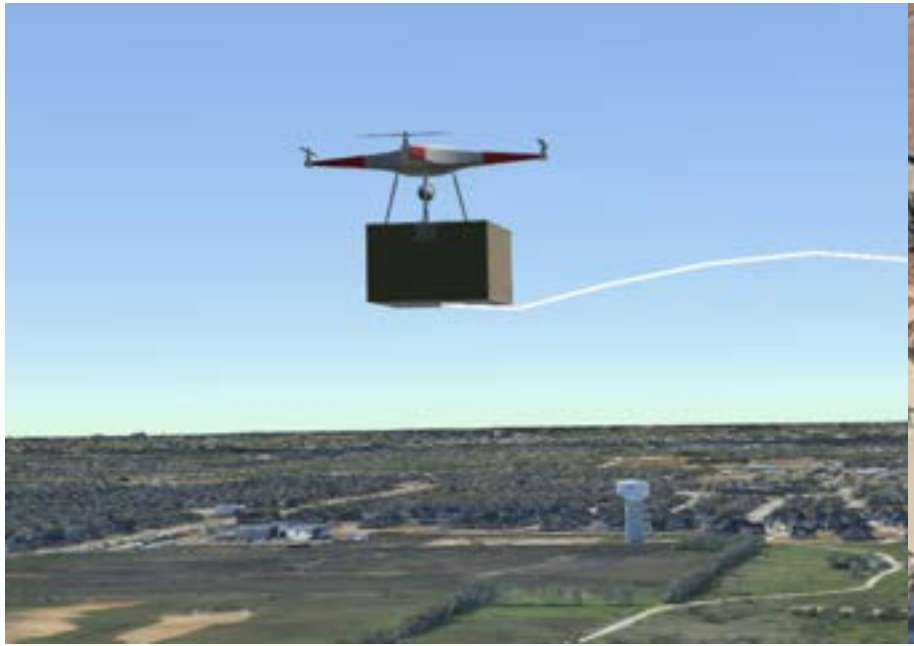
Streaming & Edge IoT

Real-time data analytics and model deployment for Federated Learning at the Edge.

Agentic AI & LLM

Building complex Agentic AI, LLM, and ML pipelines at enterprise scale.

Replay, Reason, Refly



Agenda

WHERE WE'RE GOING

1. **The paradox**
   A perfect record hiding a real cost
2. **Why undetected**
   The blind spot every dashboard shared
3. **Replay**
   One Beam YAML pipeline over every fleet's archive
4. **Reason**
   Use the pipeline to correlate and find root cause
5. **Refly**
   Replay the post-fix archive; verify it's gone.
6. **The question**
   What is your fleet quietly compensating for?

The Paradox

A FLEET THAT NEVER FAILED

A VTOL logistics fleet has flown hand-selected delivery corridors for 14 months.

Every flight clears
health checks before and after takeoff

Every recovery is clean
the autopilot stabilizes and continues

No alerts fire
nothing crosses a configured threshold

No reports are filed
operators see a perfect record

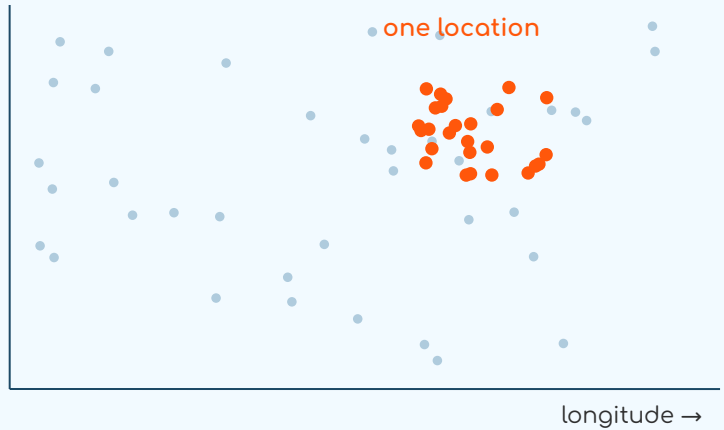
On paper, the operation is running perfectly. But what may not have been accounted for?

One Pipeline, A Different Story

1,200 LOGS · EVERY FLEET

A single Apache Beam pipeline replayed every archived flight. The dashboards and the data disagreed.

High rotor-load events cluster



One region

High-load events concentrate at a specific area along the route

One altitude band

The same vertical slice, flight after flight.

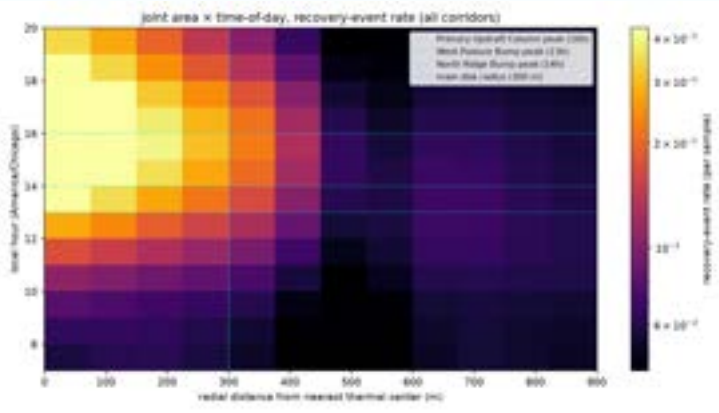
For both human and autonomous flights

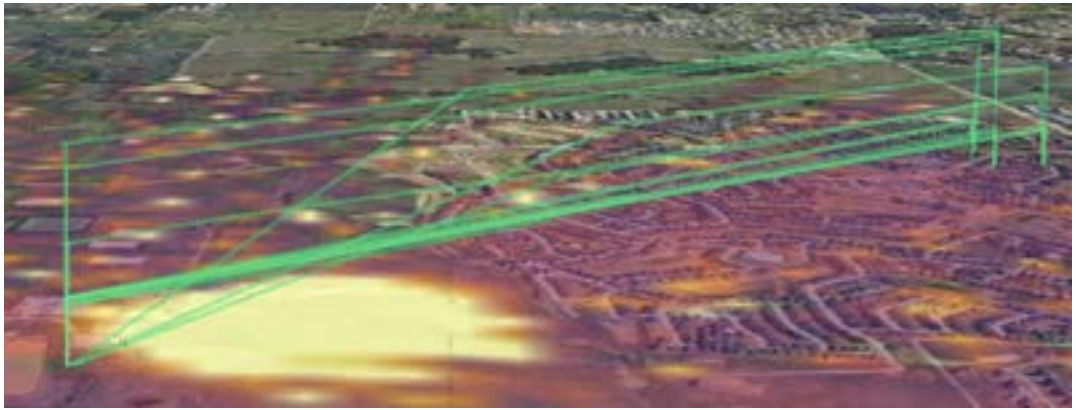
Sensor data reveals the same physical effect.

Pilots & Autopilot never complained

THE SILENT FIGHT

The pilots & autopilot were fighting thermal plumes in their corridor, which was spatially constrained.



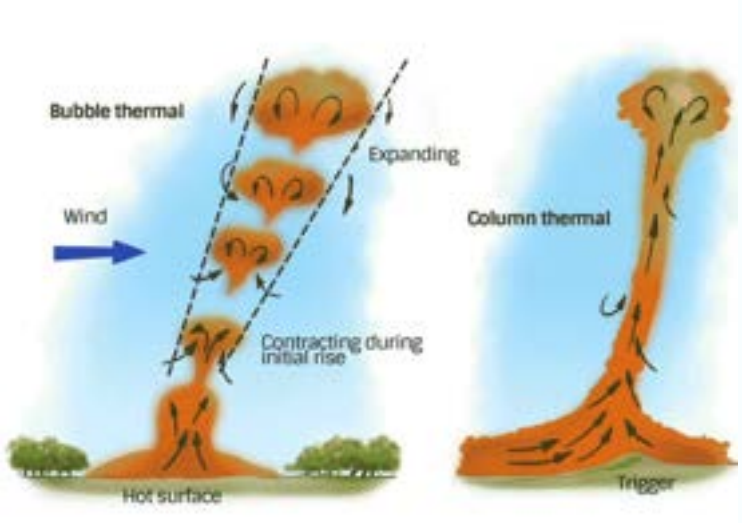


Each correction was small, clean, and individually insignificant. Cumulatively, it was not.

A terrain-induced thermal column

THE SILENT FIGHT

Hot surfaces at specific times of day yield: irregular & invisible volumes of rising air. Already spatially restricted, these insights can help inform future route planning.



The Cost That No Dashboard Surfaced

REAL, COMPOUNDING, INVISIBLE

The fight was free on any single flight. Across 14 months it added up to a measurable operational tax.

Excess battery draw

Extra rotor effort to hold position means shorter usable range per sortie.

Elevated motor wear

Repeated high-load corrections accelerate bearing and ESC fatigue.

Compounding time loss

Seconds of corrective maneuvering per flight, multiplied across every fleet.

None of it crossed a threshold. So none of it was ever reported.

Why It Was Never Detected

REASON · THE BLIND SPOT

Nothing was broken, so nothing reported. Every monitoring assumption the fleet relied on was satisfied.

Thresholds, not patterns

Dashboards alert when a value exceeds a pre-determined limit. Each event stayed inside tolerance, so no line was ever crossed.

Per-flight, not per-fleet

Health checks judge one flight at a time. The cost only appears when you aggregate the same location across hundreds of flights.

Success looks like silence

A clean recovery generates no alert and no log entry of concern. Winning the fight adhering to the intended route erased the evidence of it.

The Question Beam Lets You Ask

What has your fleet been quietly
compensating for
that your dashboards never showed you?

Beam makes this answerable.. The same logic over 14 months of history across every fleet, in one replayable pipeline.

How Beam Helps

HISTORY AT FLEET SCALE

Beam helps us as a harness in analyzing every log from every flight from every aircraft over its operational life cycle. We care about modular, repeatable methods.

Logs sit dark

Each aircraft writes a flight log for every flight. Across many fleets and over months these logs add up.

Per-fleet scripts don't compose

One-off queries or notebook analyses often answer bespoke questions and limited spans. They don't scale to every fleet, and they're never re-run the same way twice.

The replay gap: the answer is already in the archive. Now what questions can we ask?

Replay · Reason · Refly

Historical analysis informing future plans

1

Replay

STEP 1

Run one Beam YAML pipeline over months of archived logs from every fleet. Visualize historical flights and post-process artifacts and derived artifacts.

2

Reason

STEP 2

Explain why nothing ever flagged it, then use the pipeline to correlate clusters with surface hot spots, time-of-day, rotor effects, and command signals.

3

Refly

STEP 3

Apply the route fix, then replay the next flight log through the same pipeline to confirm the inefficiency is gone.

01

REPLAY

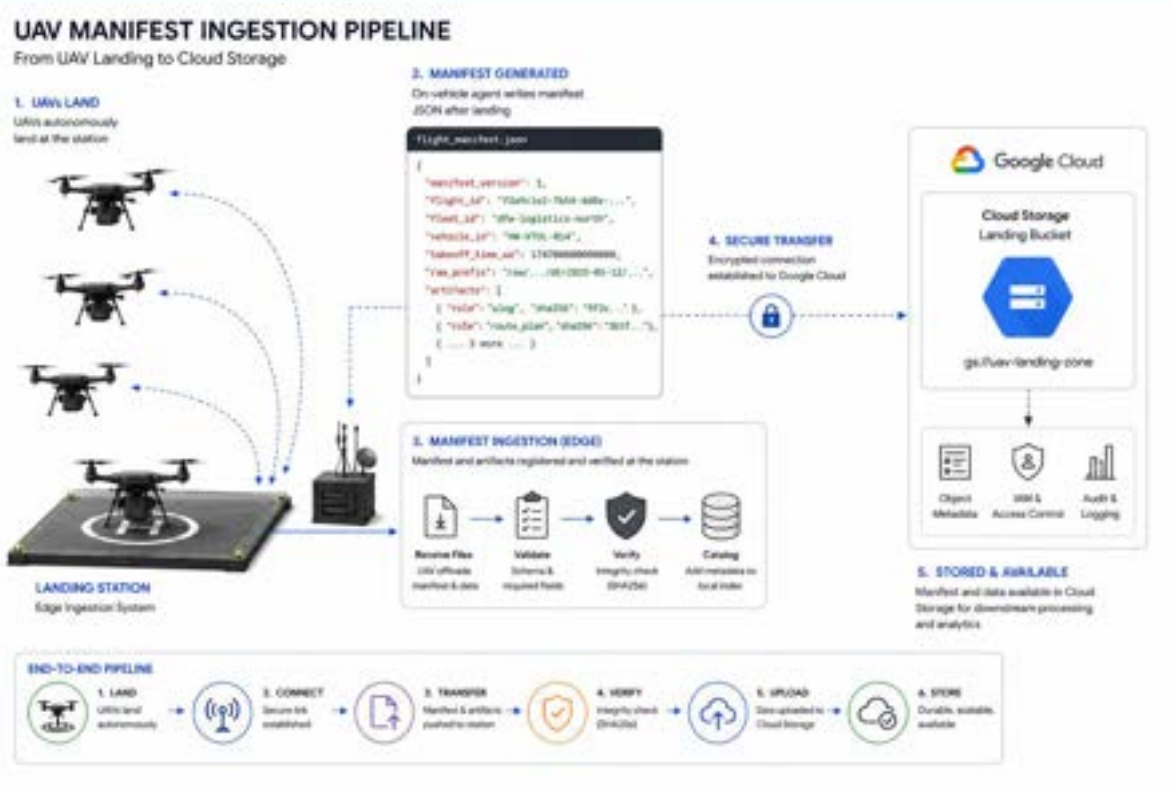
Read 14 months of archived flights and surface what no dashboard showed.

Ingest
Manifests, hashes, dead-letter

Detect
High rotor-load events along the planned route

Cluster
Correlate modalities and aggregate by location

UAV flight manifest upload



Replay the Whole Archive

ONE PIPELINE · EVERY FLEET · 14 MONTHS

Beam YAML lets one pipeline replay every archived log from every fleet — a single batch job over the full history.

No rewrite per fleet. One declarative pipeline reads them all and surfaces the patterns each fleet hid on its own.

The Dataset: Every Fleet, 14 Months

WHAT WE REPLAY

Fourteen months of archived VTOL telemetry across the whole fleet and the binary logs the aircraft already wrote to disk.

3	1,200	14 mo	50 Hz
fleets	flights archived	of operations	attitude sampling

Per-flight signals we care about

fleet id · timestamp · GPS position · altitude · attitude (roll / pitch / yaw) · attitude setpoint · battery voltage & current · motor outputs · operator commands · flight mode

Replay Architecture

HOW THE REPLAY IS WIRED

Archived logs from every fleet feed one Beam YAML graph that emits ranked clusters, diagnoses, and route deltas.

Every fleet enters the same transforms. Refly is just the same graph re-run after the route is fixed.

Why Beam YAML

THE PIPELINE AS A DECLARATION

We build the whole thing in Beam YAML as a declarative pipeline definition. Readable on a slide, runnable in production.

Declarative

Describe the transforms and their wiring, not the boilerplate. The graph reads top to bottom.

Scales to every fleet

The same YAML runs on the Direct runner for one fleet or Dataflow for the whole archive — no code change.

Reviewable

A pipeline change is a diff. Operators and analysts can read it without reading Java or Python.

```
pipeline:  type: chain
  transforms:  # each block is one step, wired in order
```

The Pipeline Skeleton

BUILD THE REPLAY · ONE FILE, TOP TO BOTTOM

Every step we are about to build lives inside one chained pipeline.

```
pipeline:
  type: chain                # each transform feeds the next
  transforms:
    - type: ReadFromJson      # Step 0  read manifests
    - type: AssignTimestamps  # Step 1  event-time
    - type: MapToFields       # Step 2  decode ulog
    - type: Filter            # Step 3  rotor-load events
    - type: Combine           # Step 4  aggregate by cell
    - type: Enrichment        # Step 5  correlate cause

providers:
  - type: python              # terrain_aspect(), on_mission_leg()
    config: {packages: [fleet_udfs]}
```

A provider makes our Python helpers callable from YAML — the same UDFs every layer reuses.

Step 0: Read the Archive

BUILD THE REPLAY · STEP 0 · INGEST

The pipeline never reads raw files first. It reads flight manifests

- one JSON per flight that declares its artifacts, timing, and hashes.

```
- type: ReadFromJson
  name: ReadFlightManifests
  config:
    path: gs://fleet/raw/*/*/dt={2024-04-01,..,2025-05-31}/*/manifest.json
    # dt= partition prunes to a date range; no full-archive scan

- type: AssignTimestamps
  name: UseTakeoffAsEventTime
  config:
    language: python
    timestamp: takeoff_time_us / 1_000_000   # micros -> seconds
```

Stamping each flight with its takeoff time means the replay is ordered by when it flew, not when we processed it.

Step 1: Fan Out & Decode

BUILD THE REPLAY · STEP 1

For each valid manifest, read its five artifacts and decode the ulog into typed, per-sample records keyed by flight_id.

```
- type: MapToFields
  name: DecodeUlog
  config:
    language: python
    fields:
      flight_id:    flight_id           # join key
      t_us:         t_us                # event-time
      lat:          lat
      lon:          lon
      alt_m_wgs84:  alt_m_wgs84
      batt_v:       battery_voltage_v
      batt_a:       battery_current_a
      rotor_load:   rotor_load_pct      # the signal
      mission_leg:  mission_leg         # which route segment
```

Five roles, one key

ulog, route_plan, command_log, flight_status, sensor_data all carry flight_id so every later join is on that one field.

rotor_load_pct is the lead

Aggregate rotor effort, 0–100. The urban-canyon anomaly lives here as a sustained spike at a fixed lat/lon.

Step 2: Detect Rotor-Load Events

BUILD THE REPLAY · STEP 2

A rotor-load event is an on-mission-leg telemetry sample whose rotor load exceeds the 78% detection threshold.

```
- type: Filter
  name: DetectRotorLoadEvents
  config:
    language: python
    keep: "rotor_load > 78.0 and mission_leg == 'on_route'"

# high effort while flying the
# planned route
```

Tied to the mission

We only keep samples taken while the aircraft is on its commanded route leg where the rotors work harder maintaining the planned path.

Key insight

Each high-load sample on its own is noise. The signal is where they cluster.

Step 3: Key by Location

BUILD THE REPLAY · STEP 3

To find a pattern, group events by where they happened
A fixed geo-cell from lat/lon plus an altitude band.

```
- type: MapToFields
  name: KeyByLocation
  config:
    language: python
    append: true
    fields:
      geo_key:
"f'{round(lat,4)},{round(lon,4)},{round(alt_m_wgs84/20)*20}'"
```

Binning lat/lon into cells turns continuous position into discrete buckets. Identical locations across different flights and fleets collapse onto one key.

Step 4: Aggregate Across Flights

BUILD THE REPLAY · STEP 4

Count events per location cell and average the rotor load there. This is where the cluster appears.

```yaml
- type: Combine
  name: AggregateByLocation
  config:
    group_by: geo_key
    combine:
      event_count:
        value: flight_id
        fn: count
      avg_rotor_load:
        value: rotor_load
        fn: mean
```

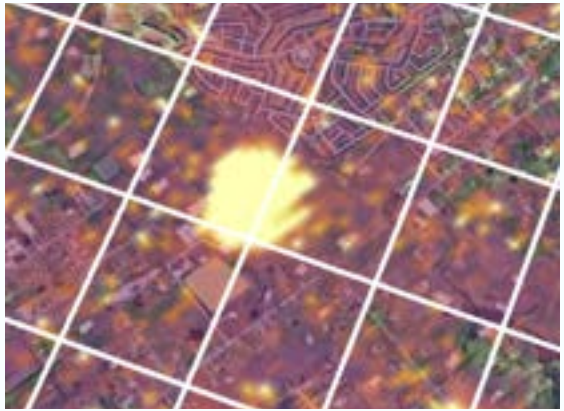
Top location cells

32.781,-96.812 / 312 events	312
32.781,-96.804 / 281 events	281
32.755,-96.796 / 36 events	36
32.799,-96.788 / 7 events	7
32.740,-96.780 / 4 events	4

The Cluster Reveals Itself

IDENTIFY AREAS OF INTEREST

Plotted, the aggregation is unmistakable: three adjacent location cells carry 97% of all rotor-load events for this delivery corridor.



recovery events per geometry key, all fleets, 14 months of replay

Running the Replay at Scale

ONE FILE, ANY SCALE

The same pipeline file replays one fleet on a laptop or every fleet on Dataflow with no code change.

```
# replay one fleet locally for development
python -m apache_beam.yaml.main --yaml_pipeline_file=fleet.yaml \
  --runner=DirectRunner \
  --jinja_variables='{"source": "gs://fleet-archive/A/"}'

# same file, every fleet at scale
python -m apache_beam.yaml.main --yaml_pipeline_file=fleet.yaml \
  --runner=DataflowRunner \
  --jinja_variables='{"source": "gs://fleet-archive/*/"}'
```

Develop against one fleet, then replay them all with the identical graph. The pipeline you tested is the pipeline you ship.

Running on Dataflow

BUILD THE REPLAY · FROM LAPTOP TO FULL ARCHIVE

The same YAML file scales to the whole fleet by changing the runner and pointing at every dt partition — nothing in the pipeline changes.

```
python -m apache_beam.yaml.main \
  --yaml_pipeline_file=fleet_replay.yaml \
  --runner=DataflowRunner \
  --project=mavencode-fleet \
  --region=us-central1 \
  --temp_location=gs://fleet/tmp \
  --max_num_workers=50 \
  --autoscaling_algorithm=THROUGHPUT_BASED
```

Develop against one fleet on the Direct runner, then promote the identical graph to Dataflow for all fourteen months. Same file, same results.

02

REASON

Use the pipeline itself to dig into root cause — correlate, don't guess.

Join
Operator commands and per-flight status by flight_id

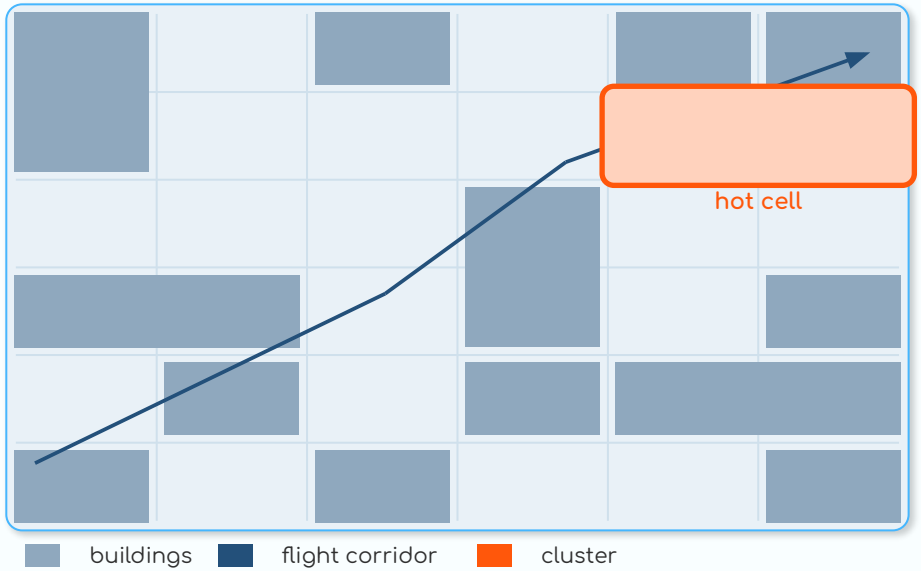
Quantify
Battery, time and wear the cluster actually costs

Correlate
Buildings and wind — the cause falls out of the data

From Cluster to Cause

REASON · HOW WE INVESTIGATE

Replay enables us to visualize our historical routes and provides a harness for analyses. Two moves, both in the same pipeline.



1

Quantify the cost

Turn the cluster into watt-hours, seconds, and wear.

2

Correlate the cause

Enrich with terrain / surface data and see the physical reason fall out of the map.

Step 7: Quantify the Cost

BUILD THE REPLAY · Step 7 · REASON

Convert the clustered events into engineering units: battery, time, and wear attributable to the rotor.

```yaml
- type: MapToFields
  name: QuantifyCost
  config:
    language: python
    append: true
    fields:
      wh_lost:  "(avg_rotor_load - 78.0) * 1.5 * (event_count * 0.0017)"
      sec_lost: "event_count * 6.0"
      wear_idx: "event_count * 0.01"
- type: MapToFields
  name: QuantifyPctRange
  config:
    language: python
    append: true
    fields:
      pct_range: "wh_lost / 500.0 * 100"
```

3.8% — **Battery**
of usable range, per affected flight

+47 s — **Flight time**
added per affected sortie

2.1× — **Motor wear**
baseline at the hot cell

Step 8: SLM Cost Analysis

BUILD LIVE · PLAIN-LANGUAGE DIAGNOSIS

The final step turns the numbers into words. A small local model reads the costed, correlated cluster and writes the analysis a human would.

```
- type: RunInference
  name: SLMCostAnalysis
  config:
    model_handler: HuggingFacePipeline
    model: Phi-3-mini-4k-instruct   # local
    prompt: |
      Cluster at {cell}, {event_count} events.
      Cost {pct_range}% range, {sec_lost}s/flight.
      Terrain {slope_aspect}, solar {solar_load}.
      Explain the likely cause + a route fix.
```

Generated analysis

"Sustained rotor load over a surface hotspot indicates a terrain-induced thermal column. Raise the corridor 15 m and shift 60 m north to clear it."

Small, local, cheap.

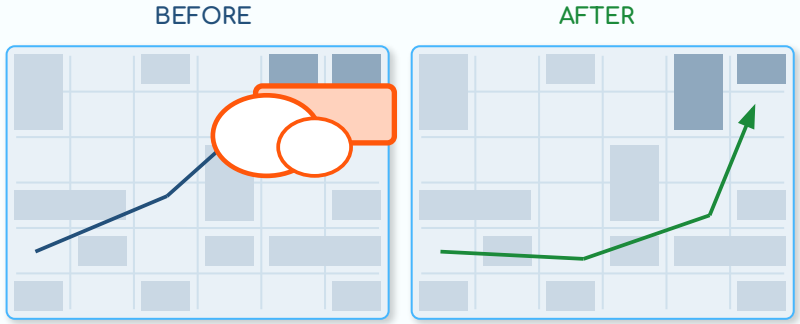
The SLM runs in-pipeline with no external API. It sees only cluster facts and returns text + a route delta

Step 8: Route Recommendation

BUILD THE REPLAY · Step 8 · REASON

Turn the diagnosis into a concrete route delta for future operations.

```
- type: MapToFields
  name: BuildRouteDelta
  config:
    language: python
    fields:
      cell: "'32.781,-96.802'"
      alt_delta_m: "+15"
      lateral_shift_m: "60"
      cause: "'thermal column effect'"
- type: WriteToPubSub
  name: PublishToPlanner
  input: BuildRouteDelta
  config:
    topic: "projects/mavencode-fleet/topics/route-deltas"
    format: JSON
```



Published to the planner. An operator approves before any aircraft flies it.

Closed loop: the pipeline that found the problem proposes the fix.

The Full Pipeline, One View

fleet.yaml

```yaml
# ============================================================
# REASON STAGE  —  cost -> operator check -> cause
#                  -> label -> SLM analysis -> route fix
#
# Input:  AggregatedByLocation (from replay_stage.yaml), one row
#         per geo cell:
#         geo_key, lat, lon, alt_m_wgs84, event_count, avg_rotor_load
# Output: route deltas published to the planner
# ============================================================

pipeline:
  type: chain
  transforms:

    # ---- Step 6 — Quantify the Cost ------------------------
    # Turn each cluster into engineering units a planner cares
    # about. Everything derives from excess_rotor_load, so the
    # numbers trace back to one measured quantity.
    - type: MapToFields
      name: QuantifyCost
      config:
        language: python
        append: true          # keep the incoming cluster fields
        fields:
          excess_rotor_load: "lambda r: max(r.avg_rotor_load - NOMINAL_LOAD, 0.0)"
          wh_lost:           "lambda r: r.excess_rotor_load * P_NOM_W * DT_HOURS"
          sec_lost:          "lambda r: r.event_count * AVG_RECOVERY_S"
          wear_idx:          "lambda r: r.event_count * WEAR_PER_EVENT"
          pct_range:         "lambda r: (r.wh_lost / PACK_WH) * 100.0"

    # ---- Step — Operator Commands --------------------------
    # Rule out a human cause. Left join keeps every cluster even
    # when no command matches -> we can see the gaps, not just
    # the hits. The finding here is the ABSENCE of commands.
    - type: Join
      name: JoinOperatorCommands
      input:
        events: QuantifyCost
        cmds:   ReadCommandLog
      config:
        type: left
        equalities:
          - events.flight_id == cmds.flight_id
          - "abs(events.t_us - cmds.t_us) < 5_000_000"   # +/- 5 s
        fields: >
          geo_key, lat, lon, alt_m_wgs84,
          event_count, avg_rotor_load,
          wh_lost, sec_lost, wear_idx, pct_range,
```

Replay Result: The Map of the Problem

REPLAY · 14 MONTHS

Run once over the archive and the unnoticed became an understood volume of airspace leading to real operational impact.

3 cells	196-227 m	3 fleets	649
keyed by cell ID	WGS84	all affected	clustered events

Diagnosis: a thermal column at a fixed region in a delivery corridor.
Cost: 3.8% range, +47s, and 2.1× motor wear on every flight through the band for 14 months, unreported.

03

REFLY

Apply the fix, then replay the next archive to prove the inefficiency is gone.

Recommend
A route delta the planner can ingest directly

Apply
Operator-approved, pushed to every affected fleet

Verify
Replay the post-fix window — the cluster vanishes

The Route Gets Adjusted

REFLY · THE FIX

An operator implements the recommended delta. The corridor is updated for every fleet that flies that cell during the impacted spans of the day.

Before

Corridor crosses cell A at 196-227m straight over the terrain inducing the thermal air column effect.

After

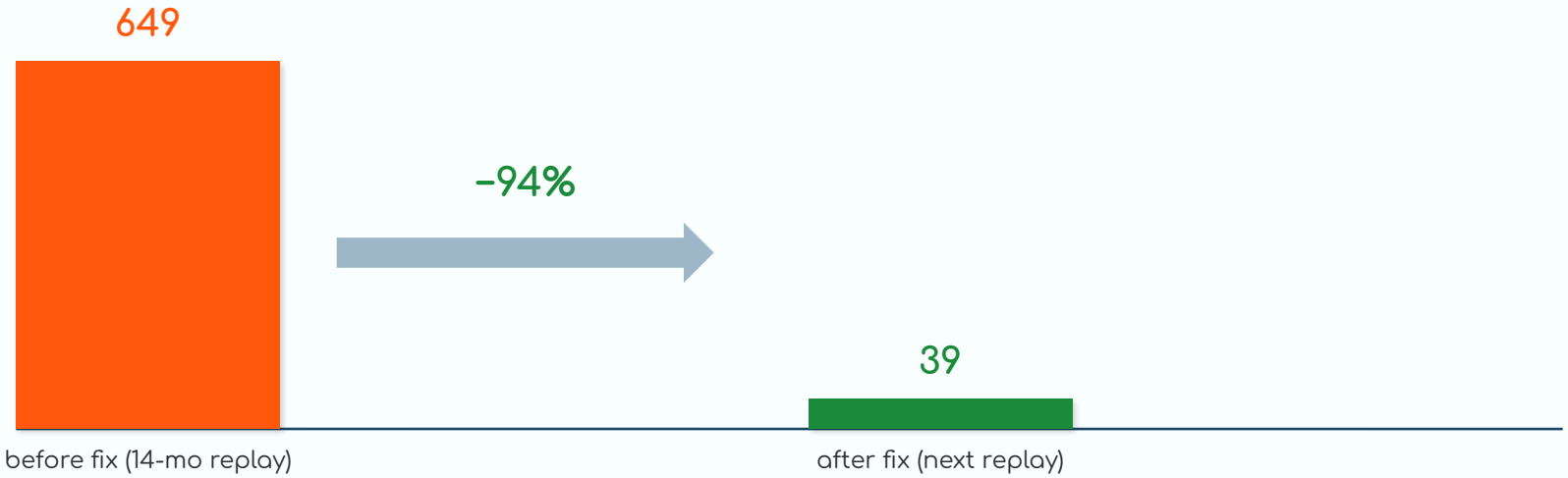
Corridor raised +15 m and shifted 60 m laterally — out of the impacted volume. This insight shared with all affected fleets.

Same flight time on paper. Materially less battery, less wear, and no silent fight. This is a change the old dashboards could never have justified, because they never saw a problem.

Refly: Replay the Fix

REFLY · REPLAY AFTER THE FIX

Once aircraft fly the corrected corridor, we replay the new archive window through the identical pipeline and measure high rotor-load events.



rotor-load events at cell A, before vs. after the fix

Refly: Replay the Fix

REFLY · per-segment energy over cell A

Destination	Before (Wh, ±IQR)	After (Wh, ±IQR)	Energy saved	Excess removed	Flights (before / after)
destination_001 — route via cell A	49.3 ±6.4	31.1 ±2.1	**18.2 Wh** ↓ (3.6% of pack)	−94%	198 / 26
destination_002 — route via cell A	47.0 ±5.9	30.4 ±2.0	**16.6 Wh** ↓ (3.3% of pack)	−93%	176 / 23
destination_003 — control · no hot cell	28.6 ±2.2	28.4 ±2.1	— (0.2 Wh)	—	152 / 21
destination_004 — control · no hot cell	31.1 ±2.4	31.0 ±2.3	— (0.1 Wh)	—	167 / 22

What Beam Made Possible

THE ENABLING PROPERTIES

Replay at fleet scale

One pipeline read every fleet's 14-month archive and surfaced a pattern no single flight or dashboard could.

Declarative in YAML

The detection logic was readable enough to review, and scaled from one fleet to every fleet with no code change.

Investigation as a pipeline

Root-cause analysis was joins and correlations in the graph, not a separate notebook or guess.

Re-runnable verification

Replaying the post-fix archive through the same graph made the result reproducible. The fix was verifiable.

The Takeaway

The operator never complained.
Your dashboards won't either.
A Beam pipeline will.

Replay your archive. Reason over the clusters. Refly the fix on the next archive.

One codebase asks what your fleet has been quietly compensating for and proves the answer on flights happening now.

Charles Adetiloye & Ian MacDonald

Thank you!

Twitter: @mavencode

https://www.linkedin.com/company/mavencode-llc

https://github.com/MavenCode

BEAM SUMMIT NYC 2026