Agentic Beam YAML for Real-Time Predictive Intelligence on Streaming Data

BEAM SUMMIT NYC 2026

By Charles Adetiloye
Agentic AI Engineering

About the Presenter

Charles Adetiloye

Cofounder & Lead AI Engineer, MavenCode

With over 20 years of experience, Charles specializes in building large-scale distributed applications and production-grade GenAI / ML platforms.

- Expertise in Agentic AI & LLM pipelines
- Consultant for major enterprise ML platforms
- Focused on high-scale distributed intelligence

About MavenCode

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

AI/ML Infrastructure
Provisioning scalable infrastructure across OnPrem and Cloud environments.

ML Platform Ops
Full-cycle development and production operationalization of ML platforms.

Streaming & Edge IoT
Data analytics and model deployment for Federated Learning at the edge.

Agentic AI & LLMs
Building large-scale pipelines for Agentic AI, LLMs, and modern ML.

Streaming data reacts. Operations need anticipation

Every team running real-time data hits the same wall. These four problems are the throughline, and Beam YAML plus agentic reasoning is how we have answered all four.

High-velocity event streams

Modern sources emit thousands of events per second across many fields, far faster than a person or a dashboard can meaningfully watch.

Problems surface too late

Fixed threshold rules fire only after something has already broken. By then the damage, the cost, or the outage is already underway.

Logic changes mean redeploys

Hardcoded pipeline logic forces a rebuild-and-redeploy cycle every time a threshold, route, or model needs to change in production.

Raw data isn't intelligence

Dashboards show what already happened. Teams need a system that reasons about what is about to happen next, and explains why.

Put the reasoning inside the pipeline

The Core Idea

Declarative

Beam YAML describes the pipeline as configuration, sources, transforms, sinks, with zero imperative code.

Agentic

LLM-powered agents run as transforms: they detect anomalies, reason about cause, and decide on alerts.

Adaptive

Parameterized templates reconfigure logic at runtime. New thresholds or models ship as config, not code.

Beam's unified batch + streaming model is the substrate that makes all three composable.

Beam YAML: Pipelines as declarative configuration

No SDK boilerplate. A pipeline is a list of transforms wired by name. The same YAML runs on Dataflow, Flink, or the Direct runner.

Composable transforms
Each step is a named, typed block, read, window, map, filter, write.

Runtime parameters
Values injected at launch with --jinja_variables; no recompile.

Runner-agnostic
One spec, many engines. Portability is a first-class property.

pipeline.yaml

```yaml
pipeline:
  type: chain
  transforms:
    - type: ReadFromPubSub
      config:
        topic: "projects/p/topics/tlm"

    - type: MapToFields
      config:
        language: python
        fields:
          temp_c: motor_temp / 10.0

    - type: WriteToBigQuery
      config:
        table: telemetry.motor

# read -> transform -> write
# zero lines of driver code
```

Beam YAML: Pipelines as declarative configuration

No SDK boilerplate. A pipeline is a list of transforms wired by name. The same YAML runs on Dataflow, Flink, or the Direct runner.

Composable transforms

Each step is a named, typed block, read, window, map, filter, write.

Runtime parameters

Values injected at launch with --jinja_variables; no recompile.

Runner-agnostic

One spec, many engines. Portability is a first-class property.

pipeline.yaml

```yaml
# launch:
#   --jinja_variables=
#     '{topic: tlm-prod, div: 10.0}'

pipeline:
  type: chain
  transforms:
    - type: ReadFromPubSub
      config:
        topic: "{{ topic }}"

    - type: MapToFields
      config:
        language: python
        fields:
          temp_c: motor_temp / {{ div }}

# same file, new values
# no rebuild, no redeploy
```

Beam YAML: Pipelines as declarative configuration

No SDK boilerplate. A pipeline is a list of transforms wired by name. The same YAML runs on Dataflow, Flink, or the Direct runner.

Composable transforms
Each step is a named, typed block, read, window, map, filter, write.

Runtime parameters
Values injected at launch with --jinja_variables; no recompile.

Runner-agnostic
One spec, many engines. Portability is a first-class property.

pipeline.yaml

```
# Direct runner (laptop)
$ python -m apache_beam.yaml \
    --pipeline=fleet.yaml \
    --runner=DirectRunner

# Dataflow (cloud, autoscaled)
$ python -m apache_beam.yaml \
    --pipeline=fleet.yaml \
    --runner=DataflowRunner

# Flink (on-prem cluster)
$ python -m apache_beam.yaml \
    --pipeline=fleet.yaml \
    --runner=FlinkRunner

# one spec -> three engines
```

Now let's ground it: a use case that stresses every part

Where this applies

Industrial IoT machine + sensor streams

Energy grids load and fault telemetry

Financial ticks high-frequency market data

UAV / VTOL telemetry our example for today

Now let's ground it: a use case that stresses every part

Where this applies

- **Industrial IoT** machine + sensor streams
- **Energy grids** load and fault telemetry
- **Financial ticks** high-frequency market data
- **UAV / VTOL telemetry** our example for today

VTOL telemetry data characteristics

Fast Dozens of subsystems stream at up to 50 Hz, real high-velocity data.

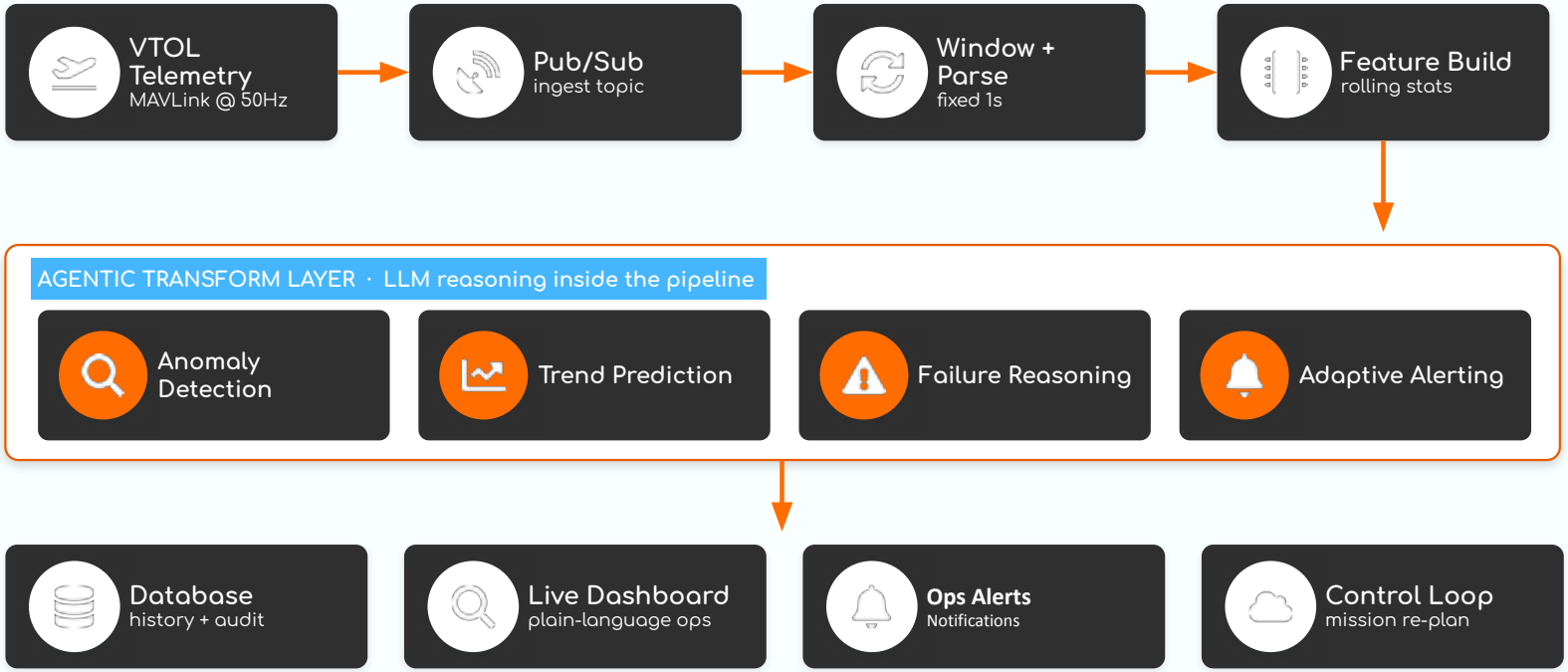
High-stakes A missed signal can cost the airframe or the mission, so anticipation genuinely matters.

Rich and varied Thermal, electrical, mechanical, and motion signals all at once, perfect for multi-signal reasoning.

Mode-dependent Hover, transition, and cruise change what normal looks like, exactly where context-aware agents shine.

From rotor to reasoning, in one stream graph

END-TO-END ARCHITECTURE



Watching a VTOL's vulnerable parts in flight

LIVE USE CASE / SCENARIOS

A VTOL transitions between hover and forward flight, the most stressful regime for its hardware. We instrument the four parts most likely to fail and feed them into the agentic pipeline.

Lift Motors & ESCs

Overheat under sustained hover load

temp, current, RPM

Battery Pack

Voltage sag during transition spikes

cell V, current, SoC

Tilt-Rotor Servos

Mechanical wear at transition angle

angle, torque, jitter

IMU / Vibration

Prop imbalance and resonance

accel RMS, gyro drift

What a single telemetry record looks like

Step1: Data Structure

Every parsed message is a flat record, timestamped at the edge and keyed by aircraft + subsystem so the pipeline can window and group it.

ts	event time, edge clock
aircraft_id	fleet key
part	motor | batt | servo | imu
motor_temp_c	°C, lift motor
current_a	amps draw
cell_v	pack voltage
vib_rms	accel RMS, g
flight_phase	hover | trans | cruise

telemetry record (JSON on the wire)

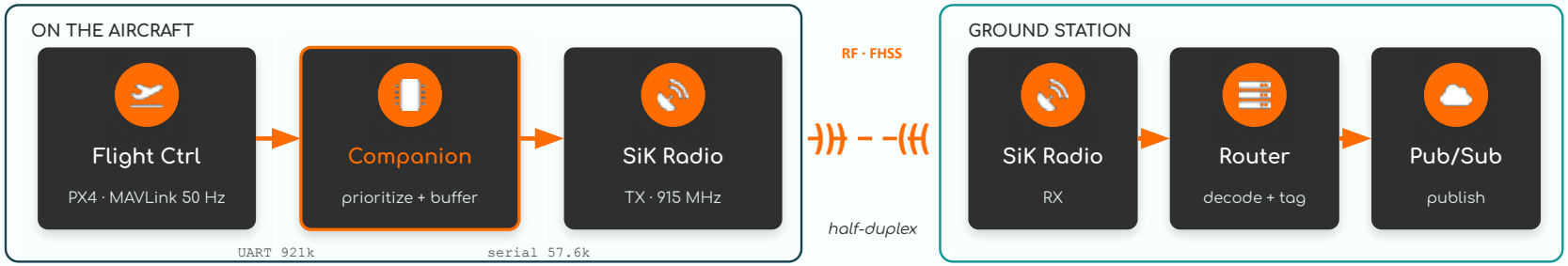
```
{
  "ts": "2026-06-09T14:03:11.480Z",
  "aircraft_id": "VTOL-07",
  "part": "motor",
  "flight_phase": "transition",
  "motor_temp_c": 86.4,
  "current_a": 42.1,
  "rpm": 7320,
  "cell_v": 3.61,
  "vib_rms": 0.94
}

# keyed by (aircraft_id, part)
# windowed on event time `ts`
```

What a single telemetry record looks like

Step1.5: Transport to Ground Station over RF

What a single telemetry record looks like

Step1.5: Transport to Ground Station over RF



1 Read

Companion ingests the full 50 Hz MAVLink stream from the flight controller over UART.

2 Prioritize

Down-rates each message type to fit the air link, shedding low-priority traffic first.

3 Transmit

Writes framed MAVLink to the radio; radios hop channels in half-duplex time slices with ECC.

4 Throttle

Radio reports txbuf + RSSI via RADIO_STATUS; the companion buffers and slows before saturation.

The air link, by the numbers

Band / air rate	915 MHz ISM · 57.6–115 kbps shared
Multiplexing	half-duplex TDM + frequency hopping
Reliability	Golay ECC + RADIO_STATUS flow control

companion push loop

```
for msg in fc.recv():              # UART, 50 Hz
    if prio(msg) >= level:         # fit air link
        radio.write(msg.pack()) # push to RF
    st = radio.status()            # RADIO_STATUS
    if st.txbuf < 20: level += 1   # shed
    elif st.txbuf > 80: level = 0 # restore
```

Read, window, and build rolling features in YAML

Step 2: Ingestion Pipeline

ingest.yaml

```yaml
- type: ReadFromPubSub
  config:
    topic: "projects/{{ project }}/topics/tlm"
    format: JSON

- type: WindowInto
  config:
    windowing: fixed
    size: {{ window_secs }}s

- type: Combine   # rolling features
  config:
    group_by: [aircraft_id, part]
    combine:
      temp_mean: mean(motor_temp_c)
      temp_max:  max(motor_temp_c)
      vib_p95:   quantile(vib_rms, 0.95)
```

What's happening

Subscribe to the live telemetry topic and decode JSON automatically.

Group events into fixed one-second windows on edge event time.

Reduce each window into rolling features the agents can reason over.

{{ project }} and {{ window_secs }} are Jinja parameters, injected at launch.

One Pattern, four specializations: BEAM YAML per part

Step 2: Per subsystem

Each subsystem filters to its own records and reduces to the features that predict its failure mode. Same transform shape, different fields.

Lift Motor

motor.yaml

```yaml
- type: Filter
  keep: part == 'motor'
- type: Combine
  combine:
    temp_max: max(temp_c)
    rpm_mean: mean(rpm)
    amp_p95:
q(current,.95)
```

Battery Pack

batt.yaml

```yaml
- type: Filter
  keep: part == 'batt'
- type: Combine
  combine:
    v_min:    min(cell_v)
    sag:      max(v)-min(v)
    draw_max: max(current)
```

Tilt Servo

servo.yaml

```yaml
- type: Filter
  keep: part == 'servo'
- type: Combine
  combine:
    ang_err:  mean(cmd-pos)
    jitter:   std(angle)
    torq_max: max(torque)
```

IMU / Vibration

imu.yaml

```yaml
- type: Filter
  keep: part == 'imu'
- type: Combine
  combine:
    rms_p95: q(accel,.95)
    drift:   mean(gyro_d)
    peak_hz: argmax(fft)
```

Composed together by Partition, every part reasons in its own lane, in parallel, on the same stream.

One template, many missions, reconfigured live

Step 3: Adaptable Pipeline

launch: same YAML, different params

```
# Test flight: relaxed, verbose
python -m apache_beam.yaml.main \
  --yaml_pipeline_file=ingestion_pipeline.yaml \
  --jinja_variables='{
    "window_secs": 5,
    "temp_limit": 95,
    "agent_model": "qwen3:8b" }'

# Search-and-rescue: tight, high-stakes
python -m apache_beam.yaml.main \
  --yaml_pipeline_file=ingestion_pipeline.yaml \
  --jinja_variables='{
    "window_secs": 1,
    "temp_limit": 80,
    "agent_model": "claude-sonnet-4-6" }'
```

Same artifact. No rebuild.

Field-tunable thresholds

Operators tighten temp_limit for a hot-day flight without touching code.

Variable window size

Trade latency for smoothness per mission, 1s for search-and-rescue, 5s for training.

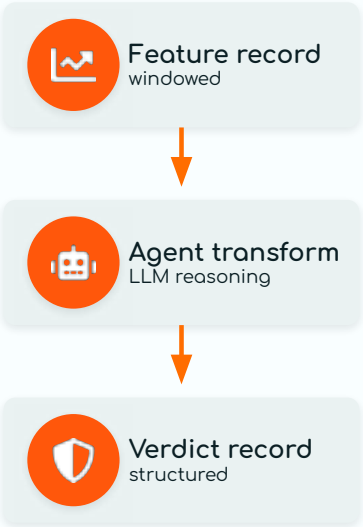
Swappable agent model

Route self hosted SLM on the edge or a frontier model in the cloud, per mission and per agent.

An LLM agent is just another Beam transform

Step 4: Agentic Reasoning

Beam YAML lets you register a custom Python transform. We wrap an LLM call so each windowed feature record flows in, and a structured reasoning verdict flows out.



agents.py: registered transform

```python
@beam_yaml.transform
def ReasonOverWindow(row, model):
    prompt = build_prompt(row)
    out = llm.call(
        model=model,
        schema=Verdict,    # typed
        input=prompt)
    return {
      **row,
      "risk": out.risk,
      "horizon_s": out.eta_s,
      "cause": out.cause,
      "action": out.action }

# in YAML:
- type: ReasonOverWindow
  config: { model: '{{ agent_model }}' }
```

Which model runs the agents? Local SLMs are a great fit

Agent Model Choice

The agent_model parameter names the model directly. A small config maps each model to its serving endpoint, from a self-hosted SLM beside the pipeline workers to a frontier model behind an API, swappable per agent, per mission, at launch.

models.yaml, serving endpoints

```yaml
agent_models:

  "qwen3:8b":  # self-hosted SLM
    provider: vllm
    host: http://llm.internal:8000
    # GPU VM beside the Beam workers

  claude-sonnet-4-6:  # frontier
    provider: api
    api_key: ${secret:llm-api-key}
    # resolved from Secret Manager,
    # never inline, never in git

# transform looks up serving info for
# whatever '{{ agent_model }}' names
```

Where the agents run

Inside the Beam pipeline, on the runner's workers. Nothing executes on the aircraft: the companion only decodes and pushes telemetry.

Self-hosted SLM

Qwen3 8B, Llama 3.2 3B, Gemma 3 4B, Phi-4-mini via vLLM or Ollama on a GPU VM in your VPC. Detection, trend, and humanize.

Cloud frontier

claude-sonnet-4-6 for the deep failure-cause correlation and the agentic YAML generation, where depth matters most.

Why SLMs fit: per-window verdicts are tiny prompts with strict JSON schemas, exactly an SLM's sweet spot, and constrained decoding guarantees the schema. Routing rule: volume goes to the SLM, depth goes to the frontier.

Four agents, four jobs, one stream

The Agentic Layer

Anomaly Detection

Flags windows that deviate from the part's learned baseline, beyond static thresholds, in full context of flight phase.

Predictive Trend

Extrapolates short-horizon trajectories: where is motor temp heading over the next 30 seconds at this climb rate?

Failure Reasoning

Correlates signals across parts to infer probable cause, rising temp plus current spike plus vibration equals bearing wear.

Adaptive Alerting

Decides whether, whom, and how urgently to alert, suppressing noise, escalating genuine pre-failure conditions.

Each agent is a stateful Beam DoFn, registered once as a YAML transform.

Beam YAML already gives you everything you need to build agents

The Agentic Layer

What you already know in Beam YAML

- Register a custom transform `providers: [agentic_transforms]`
- Window and key the stream `WindowInto + Combine, keyed by (aircraft, part)`
- Hold state per key `StateSpec: baselines, history, alert levels`
- Parameterize at launch `Jinja: {{ agent_model }}, {{ temp_limit }}`

→ *compose into agents*

the agentic pipeline, in YAML

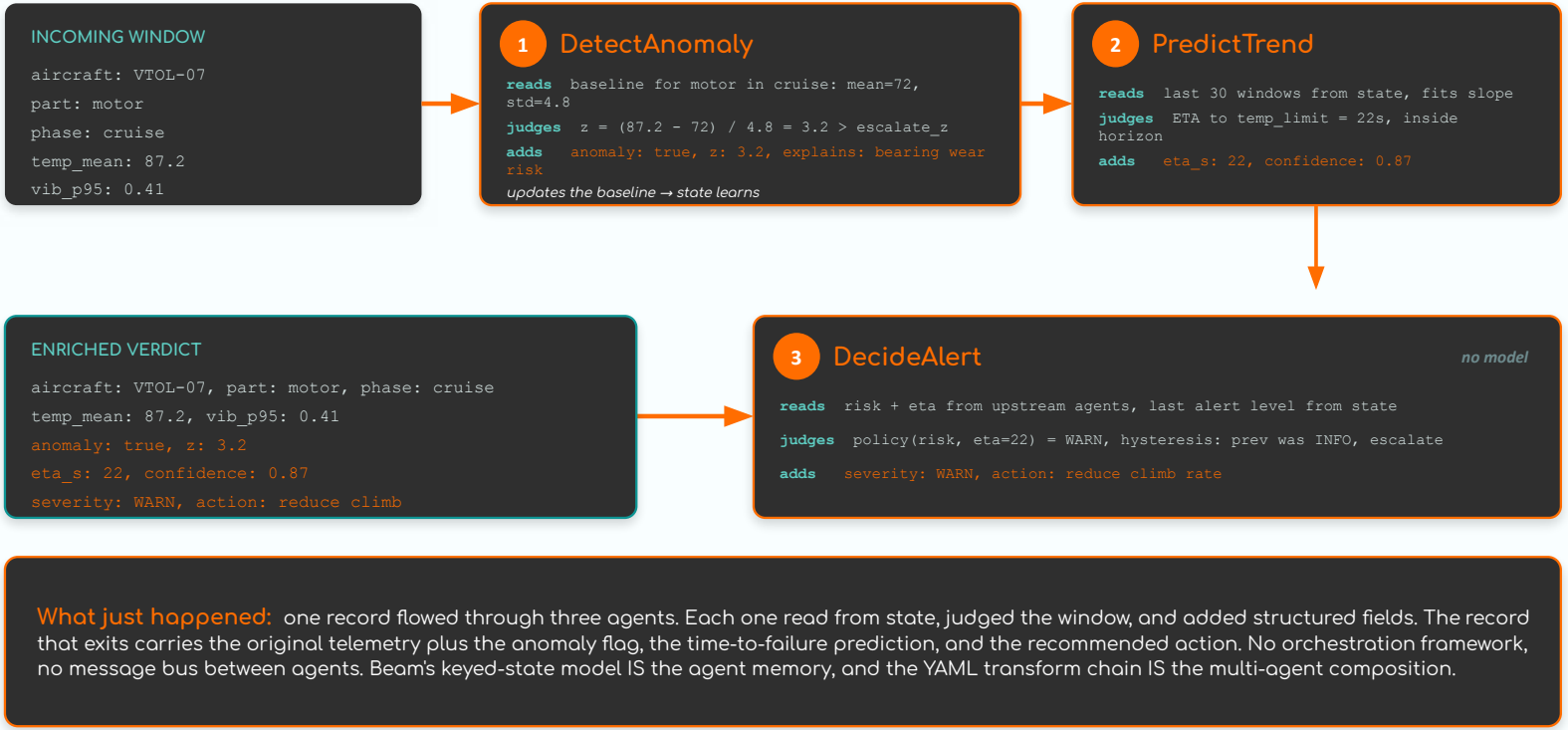
```yaml
transforms:
  - type: ReadFromPubSub
  - type: WindowInto      # known
  - type: Combine         # known
  - type: DetectAnomaly   # agent 1
  - type: PredictTrend    # agent 2
  - type: DecideAlert     # agent 3
  - type: Humanize        # agent 4
  - type: Partition       # known
```

Same YAML, same runner, same deploy. The agents are just transforms in the chain.

The insight: you don't learn a new framework to build agents for streaming data. You learn Beam YAML, and agents are a natural extension of the same primitives: transforms, state, windowing, and config. The next slides show how each agent works inside this chain.

One window of telemetry, enriched at every hop

Agents in Action



What just happened: one record flowed through three agents. Each one read from state, judged the window, and added structured fields. The record that exits carries the original telemetry plus the anomaly flag, the time-to-failure prediction, and the recommended action. No orchestration framework, no message bus between agents. Beam's keyed-state model IS the agent memory, and the YAML transform chain IS the multi-agent composition.

When a window is flagged, Escalate to the Investigator

On Demand

the verdict merges back into the stream and rides the same sinks

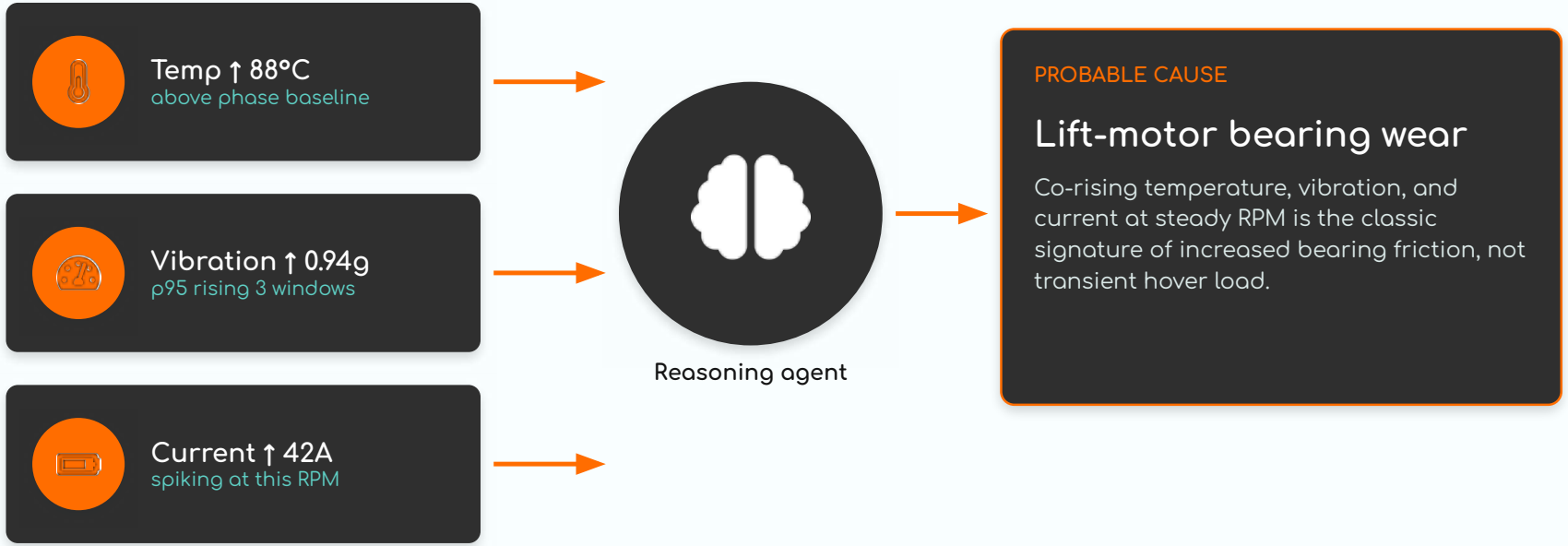
Bounded loop
max_steps and a time budget; on timeout, the tier-1 verdict stands.

Hot path never waits
Escalation is fire-and-forget over Pub/Sub; the stream keeps flowing.

Tools, audited
Every side effect is a logged tool call, replayable in a postmortem.

Correlating signals into a probable cause

The Investigator



A threshold alert would have said only: *'motor hot.'* The agent says why.

Inside the Investigator: tool calls that run the pipeline

Implementation

orchestrator.py

```
TOOLS = [
  tool("query_baselines" , bq.baselines),
  tool("fetch_history" ,  store.history),
  tool("run_projection" ,  maths.project),
  tool("trigger_pipeline" ,  # Beam as a tool
       lambda tpl, **p: launcher.run(
         template=tpl, jinja_variables=p)),
  tool("page_oncall" ,    pager.page),
]

def investigate (event):   # one flagged window
  agent = Supervisor(
    model= "claude-sonnet-4-6" ,
    tools=TOOLS,
    max_steps= 6, budget_s= 20,   # bounded
    output=CauseVerdict)    # typed, enforced
  return  agent.run(investigate_prompt(event))
```

wired in the YAML

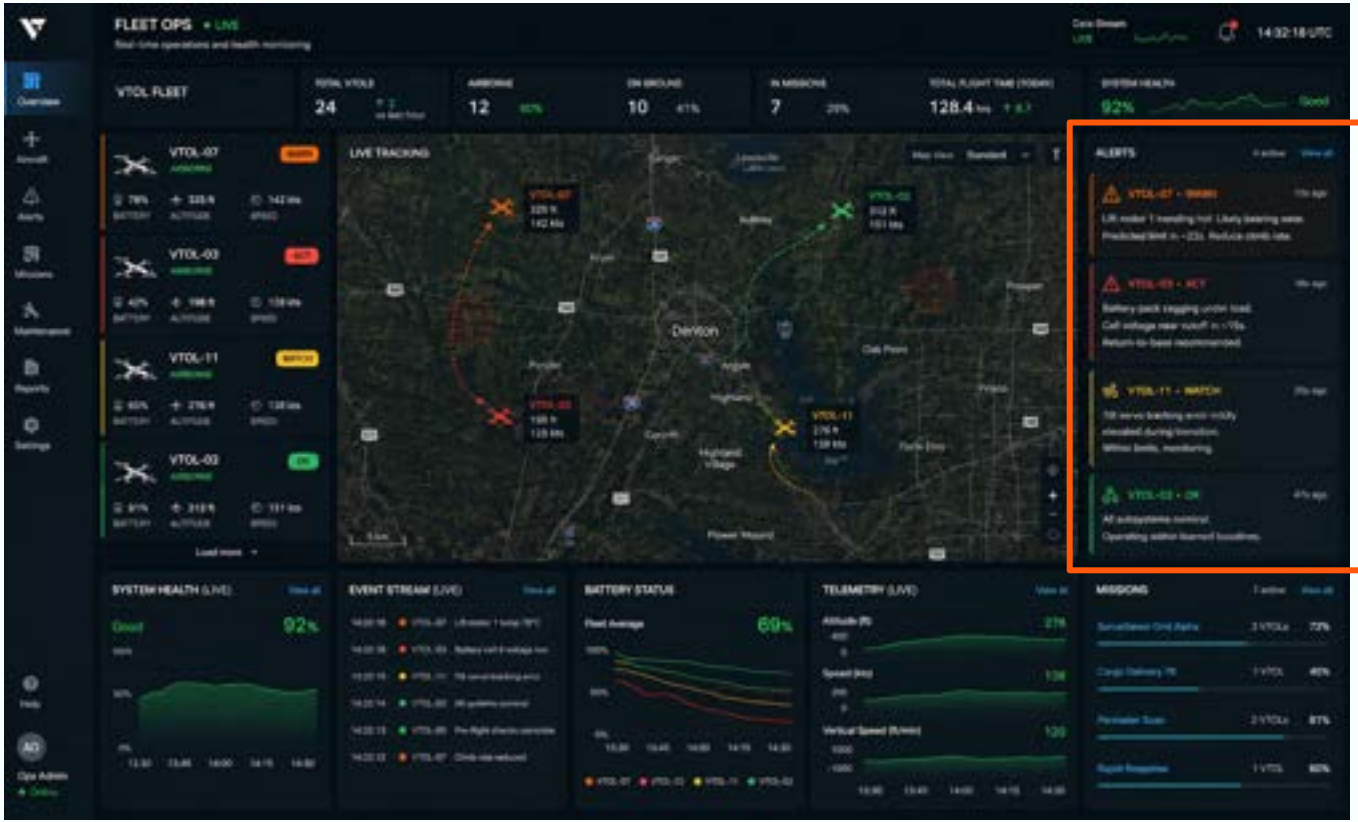
```
- type:  Partition   # flagged?
- type:  WriteToPubSub
  topic:  investigations
# orchestrator consumes, then:
- type:  ReadFromPubSub
  topic:  verdicts
```

trigger_pipeline launches a parameterized Beam YAML template, e.g. a historical replay over this airframe's logs, the agentic-engineering loop as a runtime tool.

max_steps plus a wall-clock budget bound the loop; on timeout the tier-1 verdict stands and the miss is logged.

Every tool call is logged with arguments and result, the investigation is replayable in a postmortem.

The agent explains what is happening, in plain language, on a live board



Alerts and notifications Generated By AI Agents

The complete agentic pipeline, end to end

Putting it all together

pipeline.yaml: the full chain

```yaml
transforms:
  - type: ReadFromPubSub
  - type: WindowInto  # {{window_secs}}s
  - type: Combine      # features
  - type: DetectAnomaly
  - type: PredictTrend
  - type: DecideAlert
  - type: Humanize     # verdict to prose
  - type: Partition     # by severity

sinks:
  - WriteToBigQuery  # all verdicts
  - WriteToWebSocket # ops dashboard
  - WriteToPubSub    # alerts
  - WriteToPubSub    # investigations, tier 2
  - WriteToPubSub    # control loop
```

Read it top to bottom

1. Telemetry is read, windowed, and reduced to rolling features.
2. Detect and Predict reflexes annotate each window; flagged ones escalate.
3. Alerting sets a severity; Humanize turns the verdict into prose.
4. Five sinks: history, the live board, alerts, tier-2 investigations, control.

From Apache Beam YAML to Live Execution Graph

Why this scales

- Agent calls run as parallel DoFns, the runner autoscales workers to stream volume.
- Stateful windowing means each part is reasoned over independently, in order.
- Dead-letter routing isolates a slow or failed agent call from the whole stream.

Four things to remember

Key Takeaways

Declarative pipelines are reconfigurable
Beam YAML plus Jinja parameters means logic changes ship as config, no rebuild, no redeploy, even mid-mission.

Agents belong inside the pipeline
An LLM is just a transform. Reflexes live in the stream; deep reasoning escalates to a tool-calling Investigator on demand.

Anticipation beats reaction
The result is a pipeline that predicts a failure seconds ahead, explains the cause, and can act, proven on live VTOL telemetry.

Agents can author the YAML itself
Agentic engineering closes the loop: the agent observes blind spots, generates the next config, and you audit what ships.

Charles Adetiloye

Thank you!

https://www.linkedin.com/in/charlesadetiloye/

github.com/MavenCode

BEAM SUMMIT NYC 2026