Edge-Aware Sport Sensor Pipelines: Governance, Compression, and Real-Time Impact Detection

Samuel Fraley

April 2026

Abstract

Wearable inertial measurement units (IMUs) present a fundamental design question for sport data systems: should a device transmit everything it captures, or filter on-device and transmit only what exceeds a threshold of interest? The first approach, full raw capture, maximizes auditability and analytical flexibility but carries continuous bandwidth, storage, and battery costs. The second approach, edge filtering, reduces those costs dramatically but introduces a governance gap. During periods of inactivity, the edge stream is completely silent, and silence cannot be distinguished from device failure. This project implements both approaches simultaneously on $40 of commodity hardware (ESP32 + MPU6050), producing concurrent raw and edge streams from a single session for direct comparison. Quantified results show a 97.5% bandwidth reduction and a 6× battery life improvement in edge mode, alongside 100% schema compliance across all sessions validated at 47,244 rows/sec. A schema versioning and migration system is also described, showing how data contracts can evolve without invalidating historical records. Finally, a signal processing finding with direct consequences for pipeline design is documented: threshold detection must precede any smoothing step, or borderline impact events will be silently missed.

1 Introduction

This project examines a tradeoff that sits at the center of sport sensor system design. When computation moves to the edge of a device, filtering data on-device before transmission, the cost in bandwidth and battery is obvious and easy to measure. The cost to governance and downstream analysis is less obvious and rarely quantified. The hardware used here costs approximately $40. That is not representative of a production IMU deployment, but the architectural question applies regardless of whether the device is a Catapult vest or a development board.

The core tension is between two competing requirements. A wearable sensor network at a collegiate football program might track 80 athletes across multiple daily sessions, generating millions of IMU samples per week. Transmitting the full raw signal continuously drains battery in hours and demands significant downstream infrastructure to ingest and store. The natural response is edge filtering: apply a threshold on-device and transmit only samples that exceed it. Commercial systems such as Catapult operate on this principle. The problem is that committing to edge-only transmission removes the ability to confirm device health during quiet periods, eliminates the

baseline signal required for workload normalization, and discards the audit record between flagged events. If an athlete sustains an impact that falls below the threshold, that event is unrecoverable.

The objective of this project is to implement both approaches simultaneously, so the cost of edge filtering can be measured against a concurrent raw ground truth from the same session. Governance infrastructure is built around both streams: schema validation at ingest, self-describing session metadata, and versioned data contracts. This report describes the implementation, the quantitative results, and how the architecture would extend to a production team environment.

2 System Design

2.1 Hardware

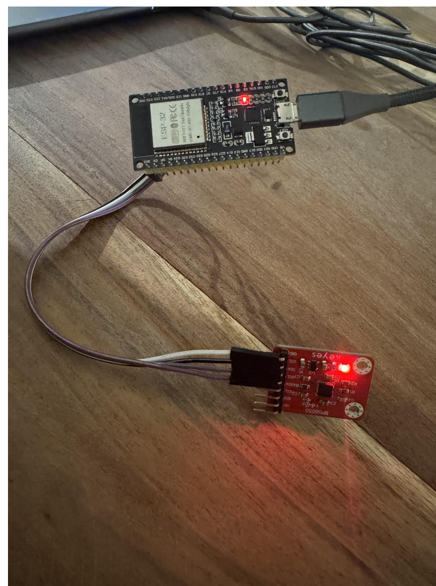

Figure 1: The physical setup: ESP32 and MPU6050 wired on a breadboard. Total component cost was approximately $40.

The hardware platform is intentionally minimal. It consists of an ESP32 microcontroller and an MPU6050 6-axis IMU, connected via I^2C (SDA $\rightarrow$ GPIO 21, SCL $\rightarrow$ GPIO 22) and powered at 3.3 V. The MPU6050 is configured with an accelerometer range of $\pm 8\,\mathrm{g}$ ($\pm 78.5\,\mathrm{m/s^2}$) and a gyroscope range of $\pm 500\,\mathrm{deg/s}$. The impact threshold of $50\,\mathrm{m/s^2}$ sits at roughly 64% of the accelerometer's full scale, which leaves headroom for large impacts without clipping. An important detail: the MPU6050 applies a 44 Hz hardware low-pass filter (`MPU6050_BAND_44_HZ`) on-chip before the firmware reads any data. Any software filtering applied later is therefore operating on a signal that has already been hardware-filtered. This is relevant to the smoothing discussion in Section 3.3.

The firmware targets a 100 Hz sample rate (`sampleDelayMs = 10 ms`), achieving 75.3 Hz in practice because I^2C read overhead extends each loop iteration beyond the nominal delay. WiFi connects on boot solely for NTP time sync via `pool.ntp.org`. After that, timestamps are derived from `millis()` offsets and WiFi plays no role during sampling. All data is transmitted over USB

serial at 115200 baud. There is no wireless data transmission in the current prototype. A production device would use a dedicated IMU with a hardware FIFO buffer, a faster bus, and on-device storage to eliminate the tethered serial connection.

The firmware operates in two modes:

Mode	Description
`RAW`	Transmits every sample unconditionally
`EDGE`	Transmits only when `accel_mag` meets or exceeds the impact threshold

Table 1: Firmware operating modes (v2 dual-stream).

The firmware computes the resultant acceleration magnitude on every sample:

$$\texttt{accel_mag} = \sqrt{a_x^2 + a_y^2 + a_z^2} \tag{1}$$

Any sample with `accel_mag` $\geq 50.0\,\mathrm{m/s^2}$ is designated a high-impact event. In the current firmware (schema version v2), both streams are emitted concurrently on every loop iteration. Each sample produces a `RAW` record, and samples that exceed the threshold additionally produce an `IMPACT` record. One distinction worth noting: `gyro_mag` does not appear in the firmware's serial output. The firmware emits twelve fields, ending with `event_flag`. The Python ingest layer derives `gyro_mag` from the transmitted `gx`, `gy`, and `gz` axes. The v1/v2 schema distinction in the Python codebase refers to the presence or absence of this derived column, not to the firmware version label.

2.2 Pipeline Architecture

Data moves from the sensor over USB serial into a Python ingest script, which validates incoming rows and writes Parquet files that DuckDB queries directly:

`MPU6050` → `ESP32` → `Serial` → `Python (ingest)` → `Parquet/JSON` → `DuckDB` → `Analysis`

Pydantic schema validation runs at the ingest boundary. Invalid rows are rejected and logged without interrupting the session. Session metadata is embedded directly in each Parquet file as columns, so queries do not require a JOIN against an external registry to identify the source of the data:

Field	Description
`session_id`	Unique session identifier
`player_id`	Player identifier (from roster system in production)
`device_id`	Hardware unit identifier
`drill_type`	Drill classification from coaching schedule

Table 2: Session metadata embedded as Parquet columns.

DuckDB registers all Parquet files as views and queries them in-place, with no ETL step required. This follows the lakehouse pattern used in enterprise sport analytics platforms, where storage and compute are decoupled and schema is applied at read time.

2.3 Data Schema

Field	**Type**	**Constraints**
`mode`	`str`	$\in$ {`RAW`, `IMPACT`}
`sample_id`	`int`	≥ 0
`utc_time`	`str`	ISO 8601
`device_ms`	`int`	≥ 0
`ax`, `ay`, `az`	`float`	m/s^2
`gx`, `gy`, `gz`	`float`	deg/s
`accel_mag`	`float`	≥ 0
`gyro_mag`	`float`	≥ 0 (v2 only)
`event_flag`	`int`	$\in \{0, 1\}$

Table 3: IMU record schema (v2). The `gyro_mag` field was added in v2 as $\sqrt{g_x^2 + g_y^2 + g_z^2}$; a migration function reconstructs it from v1 files.

Raw data lands as CSV in a local landing zone and is converted to Parquet after validation. The schema evolved across two versions during development. V1 contains the eleven transmitted fields. V2 adds `gyro_mag`, the resultant rotational magnitude, computed from the gyroscope axes that were already present in every V1 record:

$$\texttt{gyro_mag} = \sqrt{g_x^2 + g_y^2 + g_z^2} \tag{2}$$

Because the inputs were already captured in V1, the migration to V2 is lossless. No raw data is discarded and no approximation is required. The migration procedure is described in Section 5.2.

3 Signal Processing

3.1 Raw Signal Characteristics

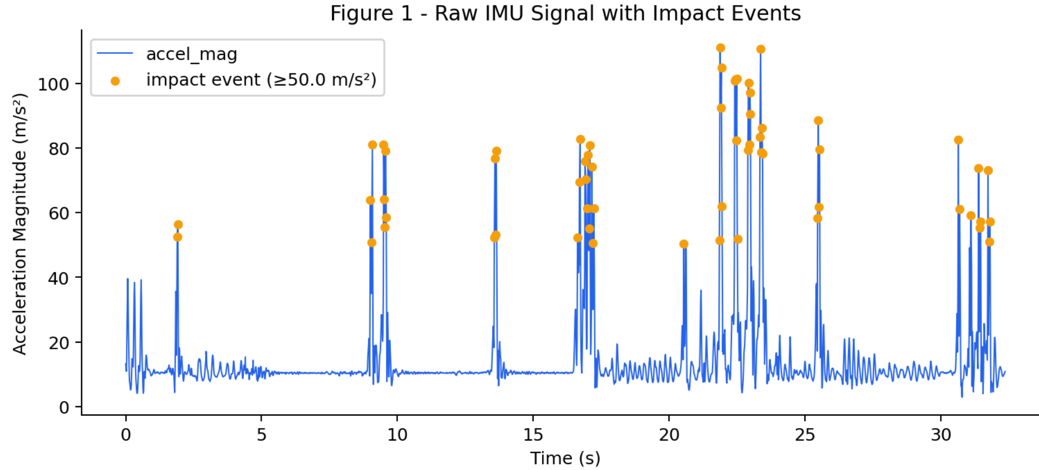

Figure 2: Raw accelerometer magnitude over a 32.4-second session. Baseline `accel_mag` rests near $9.8\,\mathrm{m/s^2}$ (gravity). Impact events appear as sharp, short-duration spikes above $50\,\mathrm{m/s^2}$, with clear rest periods between activity clusters.

Because the resultant magnitude (Equation 1) includes all three axes, it never reaches zero while the device is subject to gravity. The resting baseline sits near $9.8\,\mathrm{m/s^2}$, and impact spikes rise well above that floor. This separation is large enough that a static threshold at $50\,\mathrm{m/s^2}$ produces clean detections on this dataset. It is a favorable condition that may not hold for all sensor placements or athletes, but it is sufficient to validate the approach for this prototype.

3.2 Effect of Smoothing

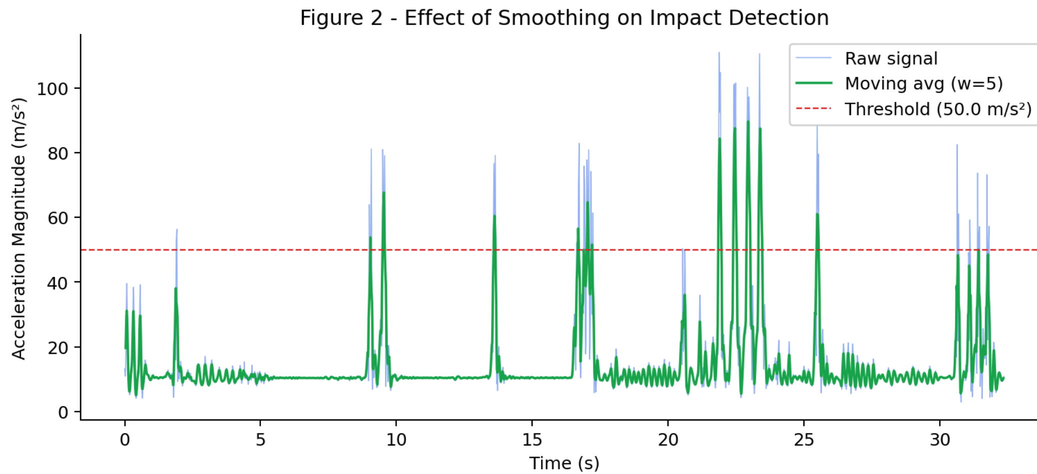

Figure 3: Moving average (window = 5) applied to the raw signal. The smoothed signal preserves low-frequency trend but attenuates high-frequency peaks; several true impact samples fall below the $50\,\mathrm{m/s^2}$ threshold after smoothing.

One important context note: the signal is already hardware-filtered before any software processing occurs. The MPU6050's on-chip 44 Hz low-pass filter removes high-frequency content before the firmware reads the data. The software moving average shown in Figure 2 is an additional layer applied to that already-filtered signal. In practice, the "raw" signal in these figures is smoother than what an unfiltered sensor would produce.

A window-5 moving average preserves the low-frequency trend but attenuates sharp, short-duration peaks. Large impacts survive the smoothing intact, but borderline events that barely exceed $50\,\mathrm{m/s^2}$ can be pulled below the threshold. This raises a practical design question: does the order of smoothing and threshold detection affect how many impact events are captured?

3.3 Smoothing Order Matters

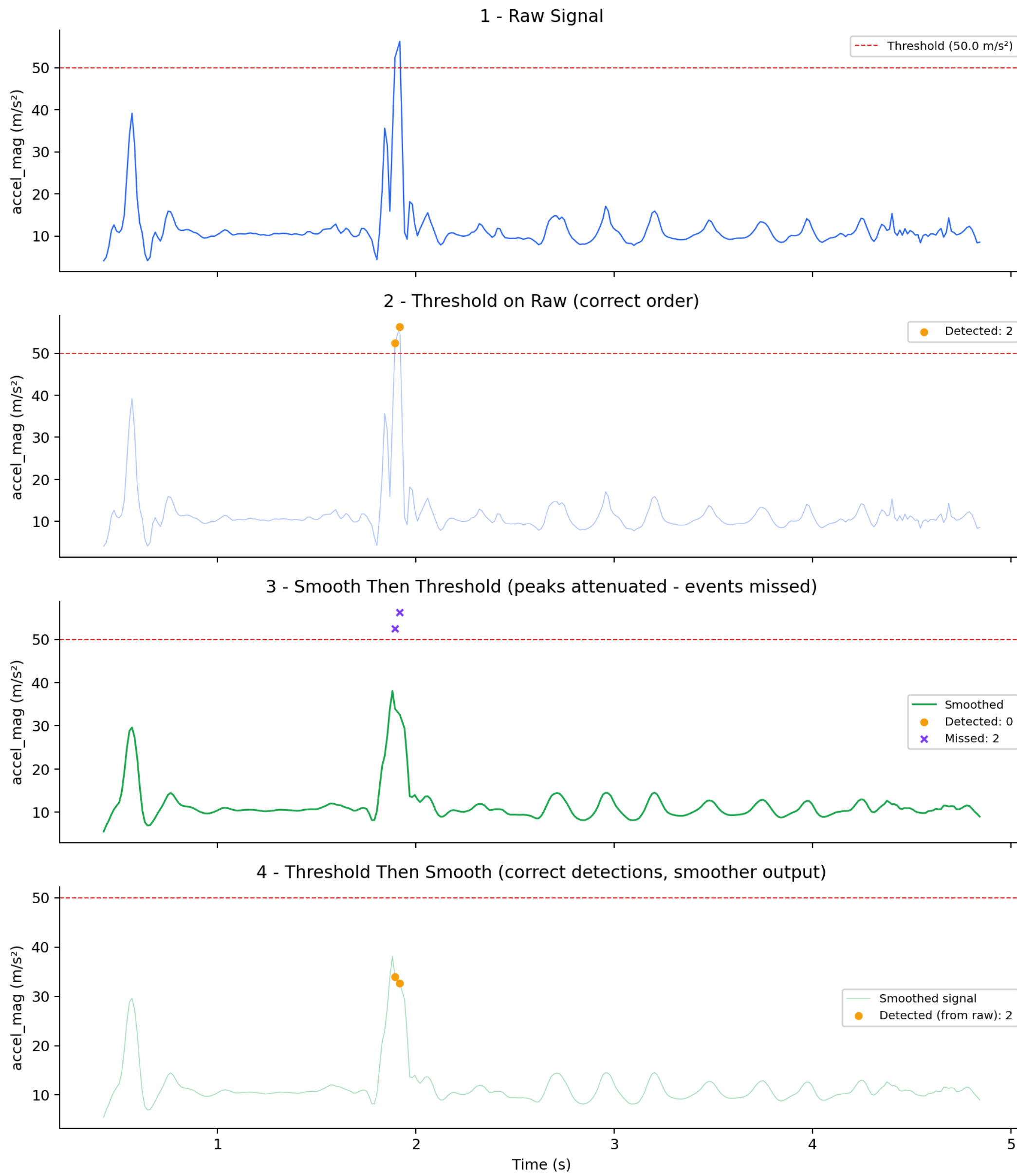

Figure 4: Four-panel comparison of smoothing order. Panels from top: (1) raw signal as ground truth, (2) threshold applied to raw, correct detections, (3) smooth then threshold, peaks attenuated and events missed (marked $\times$), (4) threshold then smooth, same detections as raw with a cleaner output curve.

Order	Outcome
Raw → threshold → smooth	Correct: all events detected
Raw → smooth → threshold	Incorrect: borderline events missed

Table 4: Smoothing order determines event recall.

The order matters. Applying smoothing before the threshold causes missed detections on borderline events. This is a category of error that is easy to overlook, because the pipeline still runs and produces output. Nothing fails visibly; events are simply absent from the record. The correct approach is to apply the threshold to the raw signal first, then smooth for visualization or downstream use. This constraint applies equally to on-device firmware logic: any filtering that precedes the threshold check on the device risks the same silent event loss.

4 Edge vs. Raw: Bandwidth and Coverage

The comparisons in this section are based on concurrent captures from the same device during the same session. The v2 firmware emits a `RAW` record on every sample and an `IMPACT` record on the same sample when the threshold fires. Both streams are real. The RAW stream is not reconstructed from the EDGE stream after the fact, and vice versa. This makes the data volume and coverage comparisons direct rather than estimated.

4.1 Bandwidth Reduction

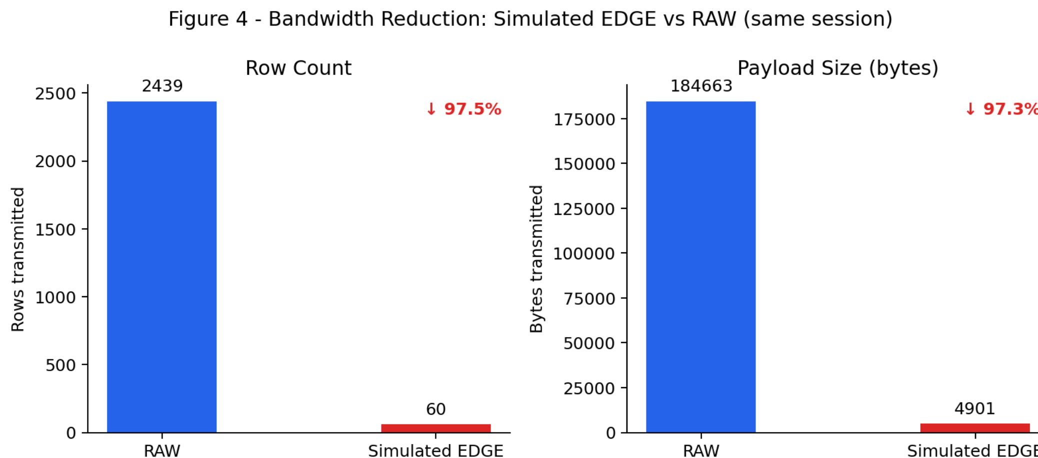

Figure 5: Row count and byte comparison between RAW and EDGE streams for the same session.

Over a 32.4-second session at 75.3 Hz, the RAW stream produces 2,439 rows while the concurrent EDGE stream produces 60 rows, a 97.54% row reduction and 97.25% byte reduction. Parquet columnar encoding provides an additional 3.3× compression over the raw CSV. At season scale (100+ sessions), this compression ratio is expected to improve further as columnar encoding amortizes schema overhead across larger row groups.

4.2 Coverage Gap

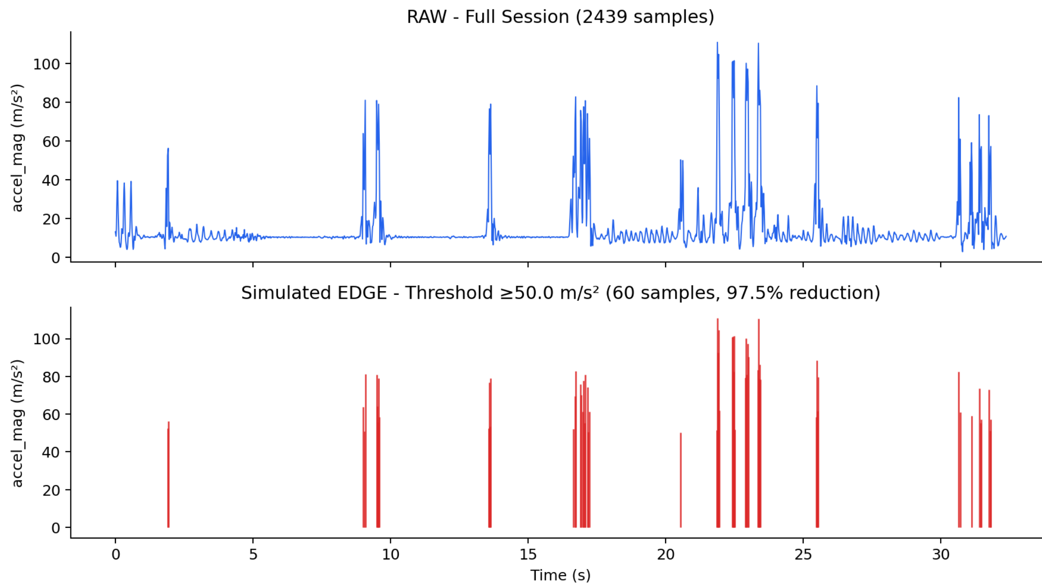

Figure 6: RAW (top) vs. EDGE (bottom) over the same session on a shared time axis. Rest windows of 10+ seconds are entirely silent in the EDGE stream.

The coverage gap is the governance cost of edge filtering that is most often underweighted. During rest windows, the EDGE stream is completely silent. There is no way to distinguish that silence from a device malfunction. For real-time alerting, this is acceptable by design. For governance purposes, such as reconstructing a game event sequence, auditing device uptime, or building cumulative load models, the silent periods represent missing data. This is not a flaw in the edge approach, but it is a constraint that must be explicitly documented in any data architecture that relies on the edge stream for anything beyond event detection.

4.3 Distribution Shift

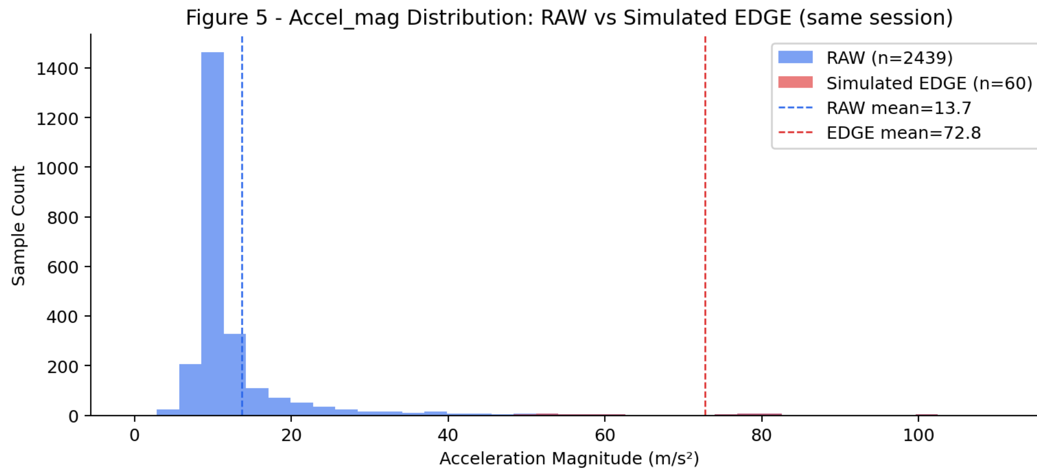

Figure 7: Acceleration magnitude distributions for RAW and EDGE streams.

The RAW distribution has a mean of $15.8\,\mathrm{m/s^2}$, which reflects the full session including rest, low-intensity movement, and impacts. The EDGE distribution has a mean of $90.2\,\mathrm{m/s^2}$ and is right-skewed, because it contains only the high-impact samples. Using EDGE data to compute a player's average acceleration load or weekly training volume would produce significantly inflated results. The EDGE stream is well-suited to answering whether an impact occurred and how large it was. It is the wrong data source for any analysis that requires the full activity spectrum.

5 Data Governance

The governance layer received the most deliberate design attention of any component in this system. The three problems it addresses are: ensuring that incoming data conforms to a known contract before it reaches storage, ensuring that stored files are self-describing without relying on external registries, and ensuring that the schema can evolve without breaking historical data. Each is addressed in turn below.

5.1 Schema Validation

Schema validation is implemented using Pydantic v2, which allows a data contract to be defined as a Python class. Each row arriving from the serial port is instantiated as an `IMURecord` object. If any field fails a type check, falls outside an allowed range, or contains an unexpected value such as an `event_flag` other than 0 or 1, Pydantic raises a validation error. The pipeline logs the bad row and continues processing. No session is interrupted and no invalid data reaches the Parquet store.

The 100% pass rate observed across all sessions is expected rather than remarkable. The firmware and the schema were designed together, so agreement between them confirms internal consistency, not an independent validation result. In a real deployment with third-party hardware or firmware

from an external vendor, a non-trivial failure rate would be expected, and this validation layer is exactly the mechanism that would surface it.

Metric	Value
Schema pass rate	100% (both streams, 4 sessions)
Validation throughput	47,244 rows/sec
Mean latency	21.2 μs/row
p95 latency	32.5 μs/row
Governance overhead vs. sampling interval	$< 0.05\%$

Table 5: Schema validation performance. The 100 ms sampling interval provides a $\sim$2000$\times$ safety margin above validation latency.

5.2 Schema Versioning and Migration

During development, it became clear that rotational magnitude should be a stored field rather than a derived quantity computed on demand in each query. Rotational acceleration is a clinically relevant variable in concussion risk assessment. A high linear impact combined with high rotational acceleration represents a qualitatively different injury risk than a high linear impact alone. Adding `gyro_mag` as a named schema field makes this dimension available to all downstream analysis and alerting without modifying how the raw gyroscope axes are stored.

The complication is that V1 data was already on disk. The migration is straightforward because the three gyroscope axes were present in every V1 record. `gyro_mag` is a derived column, so upgrading old files requires only computation, not new data collection.

The migration script (`migrate_schema.py`) follows five steps:

1. Load the V1 Parquet file and detect its schema version from the embedded `schema_version` column. Files without this column are assumed to be V1.

2. Validate the loaded DataFrame against `IMURecordV1`. Any rows that fail V1 validation are surfaced before migration proceeds.

3. Compute `gyro_mag` from the existing `gx`, `gy`, and `gz` columns using Equation 2. The inputs are present in every V1 record, so this step requires no additional data.

4. Validate the augmented DataFrame against `IMURecordV2`. The pass rate is expected to match V1 exactly, since only an additive computed field was introduced.

5. Write the migrated file as a new Parquet document with `schema_version = v2` embedded as a column. The original V1 file is left intact.

The original V1 file is preserved and the migration is fully reproducible. If the derivation of `gyro_mag` is ever questioned, the source file and the computation are both available for inspection.

Version	New field	Migration path
V1	(none)	Baseline
V2	`gyro_mag`	Computed from `gx`, `gy`, `gz` (lossless)

Table 6: Schema version history. Any V1 file can be automatically upgraded to V2 without data loss.

5.3 Session Metadata and Audit Trail

Session metadata (`player_id`, `device_id`, `drill_type`) is embedded as columns directly in each Parquet file. A file should be fully interpretable without reference to any external system. In production, `player_id` would be resolved from a roster database and `drill_type` from the coaching schedule. In this prototype, both are passed as command-line arguments at ingest time. A sidecar JSON file is also written per session to provide a human-readable audit record.

5.4 DuckDB Analytical Layer

DuckDB registers all Parquet files in the processed directory as views via a glob pattern and queries them in-place, with no database to maintain and no ETL step. Because session metadata is embedded as columns in each file, cross-session queries do not require a JOIN. A query such as returning the top five peak impacts for a given player across all sessions is a single SQL statement against the Parquet store.

Query	Time
Peak impact across all sessions	27.1 ms
Session aggregation	13.2 ms

Table 7: DuckDB query performance over the session Parquet store.

Both queries complete in under 30 ms against a four-session store on a laptop. At team scale (50+ athletes, 100+ sessions), predicate pushdown on the player and session columns is expected to maintain interactive query latency by reducing the number of scanned row groups before any numerical computation begins.

6 Benchmarks

Table 8 consolidates all quantitative results. All figures are drawn from a single 32.4-second session on v2 firmware. Validation throughput and latency are measured by `benchmark.py`, which re-validates the session CSV in isolation to exclude serial port I/O from the timing. Battery estimates use ESP32 datasheet current draw figures (240 mA active WiFi, 40 mA modem sleep) against a 1000 mAh reference cell and have not been confirmed with a physical discharge test.

Metric	**Value**
Session duration	32.4 s
Sample rate (achieved)	75.3 Hz
RAW rows	2,439
EDGE rows (concurrent, $\geq 50\,\mathrm{m/s^2}$)	60
Row reduction	97.54%
Byte reduction	97.25%
Parquet compression	3.3×
Validation throughput	47,244 rows/sec
Validation mean latency	21.2 μs/row
Validation p95 latency	32.5 μs/row
DuckDB peak query	27.1 ms
DuckDB session aggregation	13.2 ms
Battery life (RAW, ≈240 mA)	4.2 h / 1000 mAh
Battery life (EDGE, ≈40 mA)	25.0 h / 1000 mAh
Battery improvement	6.0×

Table 8: Full system benchmark summary.

7 Discussion

7.1 Recommended Deployment Architecture

In a production deployment, both streams would serve distinct and complementary roles. The EDGE stream runs continuously on-device during practice and games, transmitting only on detected impacts. It supports real-time sideline alerting with minimal battery draw. The RAW stream is buffered on-device and synced to a data lake at session end when WiFi is available, becoming the authoritative record for governance, injury review, and cumulative load modeling. Periodic raw snapshots at a fixed interval (e.g., every five minutes) could provide intermediate audit coverage without the full bandwidth cost of continuous transmission.

This architecture follows the same tiered design used in large-scale IoT systems such as AWS IoT Greengrass and Azure IoT Edge, where edge nodes filter aggressively and cloud tiers retain the complete record. In the sport context, the storage tier might be a local server in the training facility rather than a public cloud, and the latency requirement for sideline alerts is measured in seconds rather than minutes.

7.2 Limitations

Several limitations apply to this implementation. The MPU6050 at 75 Hz is below the 100–1000 Hz sample rates used in research-grade IMU applications, and well below force plate sampling rates. The impact threshold is a static value that does not adapt to player size, body position, or fatigue. Battery estimates are derived from datasheet specifications rather than physical testing. The EDGE stream in this system is produced by the same device as the RAW stream, not a separate EDGE-only

unit, so the battery comparison reflects theoretical current draw differences rather than measured values from isolated hardware configurations.

These constraints do not invalidate the architecture. A production deployment would require a higher-frequency sensor, an adaptive or learned threshold, and empirical battery characterization.

7.3 Future Work

The highest-priority extension is incorporating `gyro_mag` into the impact scoring function. The field is stored in V2 and available at query time, but the threshold logic currently uses only linear magnitude. A combined score weighting both linear and rotational acceleration would better reflect injury risk and is a natural next step given that the data is already being collected. Beyond that, upgrading to a higher-frequency sensor such as the ICM-42688-P, extending to multi-device ingestion, and connecting `player_id` to a live roster database would bring the system significantly closer to a production configuration. The Parquet and DuckDB storage layer is already structured to support all of these extensions.

8 Conclusion

This project implements a dual-path sport sensor pipeline on commodity hardware, capturing concurrent raw and edge streams from the same device to quantify the cost of edge filtering directly. Edge filtering delivers meaningful efficiency gains: a 97.5% reduction in data volume and a 6× improvement in estimated battery life. These come at a real cost. The edge stream cannot confirm device health during quiet periods and cannot support any analysis that depends on the full activity spectrum. The two streams have different appropriate uses, and a well-designed system should treat them as such.

The governance layer turned out to be as consequential as the signal processing. Typed schema validation at the ingest boundary, explicit schema versioning, lossless migration of historical files, and self-describing Parquet storage are the components that make a sensor data system reliable at scale. These are also the components most commonly absent from early-stage sport technology deployments. The Parquet and DuckDB storage pattern used here scales directly to a production environment. The sensor hardware does not, but the data architecture is the same regardless of what sits at the edge.

References

- MPU6050 Product Specification. InvenSense, 2013.
- ESP32 Technical Reference Manual. Espressif Systems, 2023.
- Catapult Sports. *OptimEye S5 Product Overview*. Catapult, 2022.
- Raasveldt, M., Mühleisen, H. DuckDB: an embeddable analytical database. *SIGMOD*, 2019.
- Pydantic v2 Documentation. `https://docs.pydantic.dev/`.

- Shi, W., et al. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5), 2016.