Everything, Everywhere, All at Once

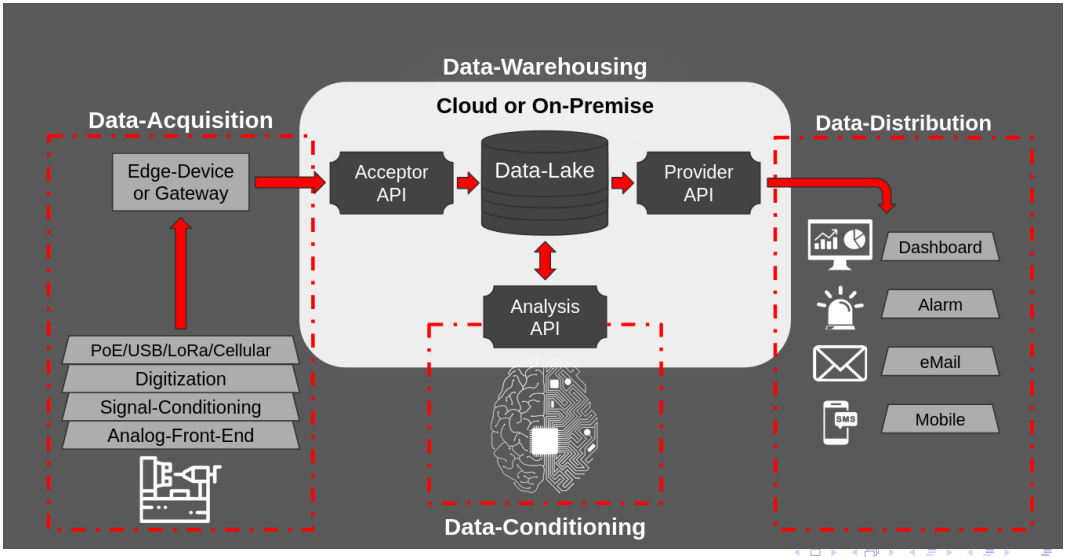
Keynote on Automated Data Collection

Stephan Bökelmann

PDSC4k – Practical Data Science Conference

2026

The Full Journey at a Glance



Agenda

1. Data Acquisition
2. Data Warehousing
3. Data Conditioning
4. Data Distribution
5. The Diagram as a Checklist

Data Acquisition: Where the Physical World Meets Software

The stack from the bottom up:

1. **Analog Front-End** – amplification, filtering, impedance
2. **Signal Conditioning** – offset, scaling, protection
3. **Digitization** – ADC, sampling rate, resolution
4. **Transport** – PoE / USB / LoRa / Cellular
5. **Gateway** – passive network bridge (modem, LoRa base station)
6. **Edge Compute** – first intelligent software node

Garbage-in Problem

Errors in the lower layers are **not** fixable by later software. A miscalibrated sensor delivers plausible but wrong data.

Analog Front-End and Signal Conditioning

What actually happens here — and why data scientists should care:

- **Aliasing**: Sampling rate must be $\geq 2\times$ the highest signal frequency (Nyquist)
- **Quantisation noise**: 12-bit ADC $\Rightarrow \approx 0.024\,\%$ resolution — is that enough?
- **Offset drift**: Temperature-dependent shift of the zero point $\Rightarrow$ systematic error
- **Electromagnetic interference**: Differential measurement, shielding, twisted pair

Takeaway

Talk to the hardware engineers before you train your model.

Edge Compute: Gateway and Compute Node in One

Horus Edge Compute Platform (custom mainboard, SBC-agnostic):

- Integrated router with **two separate ETH interfaces** (WAN / LAN isolation)
- **7 PoE LAN ports** – sensors are powered directly
- **48 V DC** supply, industrial grade
- Sensor discovery via `slook` (github: dominicpoeschko) through the Peripheral Access Daemon
- Calibration & cleaning **locally**, before data travels over WAN to the Acceptor API
- Currently in development: variant with **NVIDIA Jetson** module for inference directly at the edge

Why Compute Belongs at the Edge

Our Architectural Pattern

Sensors $\xrightarrow{\text{PoE/LAN}}$ Edge Compute $\xrightarrow{\text{WAN}}$ Acceptor API

What the Edge Compute does **before** the WAN:

1. **Aggregation** – bundle data from any number of channels (7 PoE ports, behind them any number of switches)
2. **Calibration** – apply device-specific correction factors
3. **Cleaning** – outliers, gaps, encoding errors
4. **Buffering** – store-and-forward on WAN failure

Rule of Thumb

Anything that can be **decided before** the WAN belongs at the edge – bandwidth is expensive, compute time at the edge is free.

Transport Protocols Compared

Wired

- **PoE**: power + data, simple, reliable
- **USB**: short distance, plug-and-play
- **Modbus RTU**: industrial standard, robust

Wireless

- **LoRa**: long range, low bandwidth
- **Cellular (LTE-M)**: everywhere, but costly
- **BLE**: short range, battery-friendly

The Forgotten Question

What happens to the data when the WAN connection is down for 3 hours? Store-and-forward on the **Edge Compute** is not optional – a passive gateway buffers nothing.

Starlink: Satellite Internet as a WAN Option

Starlink addresses the last blind spot in connectivity — **sites without infrastructure**:

- Latency currently $\approx$ 20–60 ms (Low-Earth-Orbit)
- Bandwidth $\approx$ 50–200 Mbit/s down
- **Starlink for IoT**: flat API programme for machine-to-machine communication, optimised for continuous operation with small packets
- Failover option alongside LTE-M for critical sites
- Hardware: compact flat-panel dish, weatherproof

When Relevant?

- Mines, offshore, construction sites
- Large-scale agriculture
- Temporary measurement campaigns
- Backup when cellular fails

Caution

Data protection: check Starlink server locations (GDPR).

Data Warehousing: Acceptor API → Data Lake

The Acceptor API is the first software gate:

- Authentication and authorisation
- Schema validation (mandatory fields, types)
- Rate limiting against flooding
- Idempotency keys enforced
- Writing into the Data Lake (append-only)

Design Principle

The Acceptor API **rejects or accepts** – it does not transform. Transformation belongs in Data Conditioning.

What Actually Belongs in the Lake?

Should Go In

- **Calibrated measurements** – pre-processed at the edge
- **Device metadata** – ID, firmware, calibration state
- **Heartbeats** – proof that a device is silent

Should **Not** Go In

- Polling artefacts without a measurement value
- Duplicates from retries (filter, do not store)
- Internal diagnostic data from the Edge Compute

Rule of Thumb

Everything required to reconstruct the state of the world **at the exact moment of measurement** – nothing more, nothing less.

Practice: Ceph as a Real Cloud Solution – with THGA

Philipp Lehmann and I, together with the **Technische Hochschule Georg Agricola (THGA)**, have built a complete Ceph-based cloud infrastructure – no longer a proof of concept, but **running in production**:

- Currently **100 TB usable** – and growing
- Real failure cases produced and analysed
- Paper in preparation

Philipp is here today – talk to him, he can tell you more.

IPFS as Storage Backend Behind the Acceptor

Architecture:

1. Acceptor writes data → **IPFS node**
2. IPFS returns a **CID** (Content Identifier = hash)
3. CID + metadata → **On-Premise Metadata DB**
4. **FastAPI catalogue** returns only metadata & CID
5. Consumer fetches the data themselves by CID from IPFS

Why IPFS?

- CID is a checksum – no silent corruption
- Immutability by design
- Deduplication for free
- Storage location interchangeable (Ceph, S3, cluster)

Separation

Metadata stays on-premise or in a separate cloud – never together with the data.

Cloud vs. On-Premise: A Real Decision

Cloud

- Elastic scaling
- No hardware operations
- Pay-per-use (watch out for egress costs!)
- GDPR: pay attention to server location

On-Premise

- Data sovereignty
- Predictable fixed costs
- Air gap for critical systems
- Internal operational effort

The Hybrid Reality

Acquisition on-premise, warehousing in the cloud – this is the most common compromise in industry.

Backpressure: What Happens When Everyone Sends at Once?

The Scenario

50 Edge Computes send their buffered data simultaneously after a WAN outage – the Acceptor API gets flooded.

Throttling (Acceptor):

- **Token Bucket** – throughput per client
- **Circuit Breaker** – 503 on overload
- **Sliding Window** – bursts OK, sustained fire not

Scaling (Infrastructure):

- **k8s HPA** – new Acceptor pods under load
- **Load Balancer** – distributes across pod pool
- Design stateless – every pod is equal

Buffering (Edge Compute):

- 503 $\Rightarrow$ backoff + jitter
- Local buffer waits
- Fresh data first

Core Idea

Throttling and scaling are not mutually exclusive – together they produce a robust system.

Data Conditioning: Where Data Learns to Think

The bidirectional connection in the diagram is not an accident:

Lake → Analysis API

- Read raw data
- Clean, normalise
- Feature engineering
- Model inference

Analysis API → Lake

- Write derived features back
- Annotations, labels
- Model outputs as new datasets
- Aggregated views for distribution

Key Idea

Conditioning **writes back** – the lake grows through analysis.

Analysis via a FastAPI Data Catalogue

Instead of delivering data, the catalogue returns only **metadata**:

- Which datasets exist?
- CID / storage location in IPFS
- Time range, device, calibration state
- Quality status (validated / not validated)

The consumer decides what to load themselves – the catalogue never transports payload data.

Advantages

- No central bottleneck
- Reproducibility: CID = exact data version
- Access control at the metadata level
- FastAPI: fast, type-safe, OpenAPI for free

```
GET
/datasets?device=T42&from=2026-01
 {cid, location, quality}
```

Orchestrating the Conditioning Pipelines

The orchestrator runs on the **analysis machine** – not in the warehouse. It queries the FastAPI catalogue for CIDs, loads data from IPFS, and writes results back.

What orchestrators provide:

- Cron-like scheduling with dependency graphs (DAGs)
- Automatic retry with configurable policies
- Backfill: re-process missing time slices later
- Alerting on failure, SLA violation
- Audit log for every pipeline run

Tool	Strength
Apache Airflow	Mature, large community, complex
Prefect	Python-native, easy to start
dbt	SQL-focused, transformation layer

Data Distribution: Who Gets What, When, How?

The Provider API is the **contractual boundary** to the outside world:

- Versioning (v1, v2 – never breaking)
- **Keycloak** – OIDC / OAuth2, role-based, SSO-capable
- Rate limiting per consumer
- Documentation (OpenAPI / AsyncAPI)

Four Consumer Types

- Dashboard (Pull, low latency)
- Alarm (Push, real-time)
- Email (asynchronous, batched)
- Mobile (Push notification)

Provider API: FastAPI or Crow/C++ – and What Does the SLA Cost?

Technology Choice by Load:

	FastAPI	Crow (C++)
Setup	hours	days
Latency	∼5 ms	<1 ms
Throughput	∼10k req/s	>100k req/s
Operations	easy	complex

Rule of thumb: below ∼20k req/s FastAPI is the right choice – above that, C++ becomes worthwhile.

What the SLA Actually Means:

- **Availability**: e.g. 99.9 % = max. 8.7 h downtime/year – who pays below that?
- **Latency**: p99 < 200 ms – not the average, but the worst-case percentile
- **Freshness**: data at most X minutes old

k8s as Leverage

HPA scales Provider pods under load – SLA targets are maintained through scaling, not through over-provisioning.

Latency and Format Requirements per Consumer

Consumer	Latency	Format	Mechanism	Priority
Dashboard	$< 1\,s$	JSON / REST	Polling / WebSocket	Medium
Alarm	$< 100\,ms$	Minimal	Webhook / MQTT	Critical
Email	Minutes	HTML / Text	Queue	Low
Mobile	$< 5\,s$	FCM / APNs	Push notification	High

Important

A single API endpoint for all consumers is an anti-pattern. Latency requirements are too different.

Alerting Done Right

Common mistakes in alarm systems:

- **Alert fatigue**: too many alarms $\Rightarrow$ all are ignored
- **No hysteresis**: threshold flapping produces alarm storms
- **Wrong recipient**: alarm goes to everyone, nobody feels responsible
- **No context**: alarm without value, trend, or action instruction

A Good Alarm

"Sensor T-42 has exceeded 85 C for 3 minutes (currently 91 C, trend: rising). Responsible: On-Call Mechanic. Runbook: `/wiki/T42`"

Separation of Responsibilities: Alarm, Analysis, Warehouse

Anti-Pattern: Everything on One Machine

When warehouse, analysis, and alerting share the same infrastructure, an outage takes down all three at once – exactly when alarms matter most.

Three Separate Systems:

- **Warehouse** – Acceptor, Lake, Provider API
- **Analysis** – Conditioning, FastAPI catalogue, ML pipelines
- **Alarm Service** – independent machine, sends email / SMS / push independently

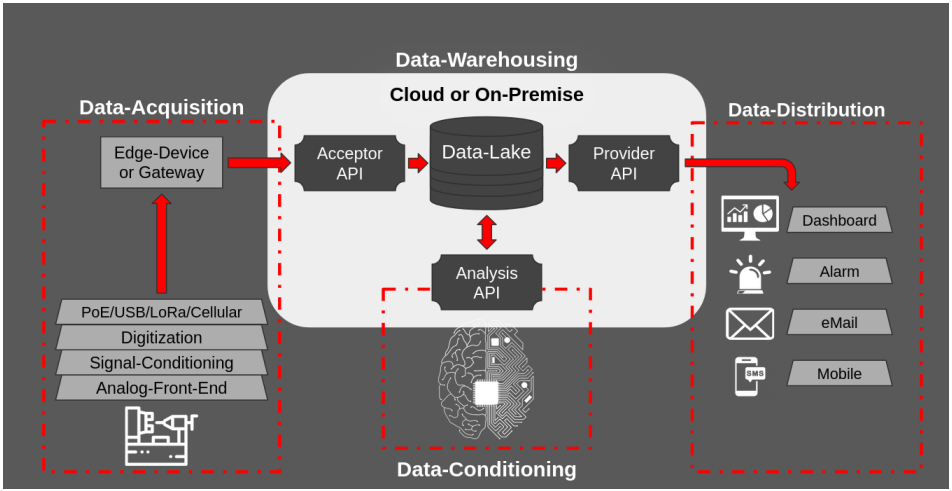
Trade-off

Higher initial setup effort –
but every component is individually maintainable, scalable, and replaceable.

Meta-Monitoring

Uptime Kuma (or similar) monitors the warehouse from the outside – monitoring the monitoring. Who watches the watcher?

Every Box Is a Responsibility Boundary



Who **owns** which box in your organisation?

Checklist: Is Your Pipeline Production-Ready?

Data Acquisition

- ☐ Calibration documented?
- ☐ Store-and-forward at the edge?
- ☐ Connection loss tested?

Data Warehousing

- ☐ Acceptor idempotent?
- ☐ Raw data stored immutably?
- ☐ `last_write_timestamp` monitored?

Data Conditioning

- ☐ Quality gates before every transformation?
- ☐ Backfill strategy documented?
- ☐ Derived data versioned?

Data Distribution

- ☐ Provider API versioned?
- ☐ Alarm hysteresis configured?
- ☐ Consumer latency tested?

Conclusion: Three Sentences to Take Away

1. **Collecting data is engineering** – not a script that runs somewhere and hopes.
2. **Quality is created at the source** – whatever the analog front-end corrupts, no model will save.
3. **Every box needs an owner** – the largest data pipelines fail on unclear responsibility, not on technology.

Your turn.

Build pipelines that don't lie.
Enjoy PDSC4k – and have a good exchange!

Stephan Bökelmann
stephan.boekelmann@gruppe.ai

github.com/maxclerkwell/pdsc4k-talk

Collaboration: www.skAInet.io
Blog: maxclerkwell.github.io