Streaming MySQL Changes to ClickHouse

Designing an End-to-End CDC Pipeline

Javier Zon · Founder, ScaleDB

Javier Zon

Founder, ScaleDB

ENGINEERING

- **15+ YEARS**
  on MySQL, distributed systems, and real-time data infrastructure
- Former
  PERCONA REMOTE DBA
  — production databases at scale

OFF THE CLOCK

- Husband & father of two
- Helping communities understand AI
- Taekwondo instructor
- Advanced Open Water diver

Still occasionally surprised by what people run directly against production MySQL.

scaledb.io · linkedin.com/in/jtomaszon · github.com/jtomaszon

Analytics Was Killing the Source Database

- BI + dashboards ran against MySQL read replicas — slow, fragile, contended
- Billions of events; cross-domain joins (orders × contacts × billing) timed out
- We needed **fresh** analytics without touching production write paths
- Goal: real-time CDC into a columnar store built for analytics

What We're Actually Moving

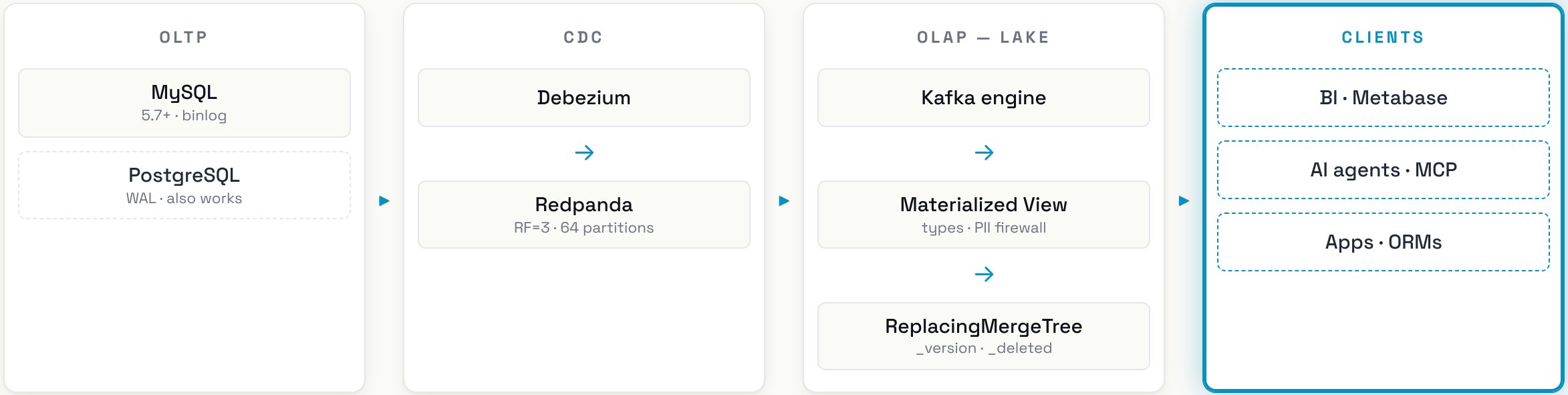
80B+
Events in ClickHouse

35TB
Analytical data

64
Redpanda partitions (RF=3)

~$4k
Full production lake cost/mo

The End-to-End Pipeline



Loose coupling — Redpanda buffers so ClickHouse downtime never loses data.

Four Decisions That Shaped Everything

01 · ReplacingMergeTree(`_version`) Last-writer-wins dedup · `_version = updated_at`

02 · Soft Deletes Deletes become a `_deleted` flag, not row removal

03 · Read-Only & PII-Safe by Construction Sensitive columns never land in analytics tables

04 · Buffer, Don't Couple Redpanda absorbs spikes and outages

Capturing Change Without Re-Reading MySQL

- One Debezium connector **per domain** (orders, contacts, products, …)
- `snapshot.mode: schema_only` → start at current binlog, never resnapshot history
- Redpanda (Kafka API): **3 nodes, RF=3, 64 partitions**, topic per table
- Why Redpanda: no ZooKeeper, simpler ops, NVMe nodes

The connector config is short — six settings do the real work.

Capturing Change Without Re-Reading MySQL

— the connector config

```
{
  "connector.class": "io.debezium.connector.mysql.MySqlConnector",
  "database.include.list": "app_production",
  "table.include.list":    "app_production.orders,app_production.orders_invoices,…",
  "snapshot.mode":         "schema_only",
  "decimal.handling.mode": "double",
  "time.precision.mode":   "connect",
  "transforms":            "unwrap",
  "transforms.unwrap.type": "io.debezium.transforms.ExtractNewRecordState",
  "transforms.unwrap.delete.handling.mode": "rewrite",
  "transforms.unwrap.drop.tombstones":      "true",
  "binlog.buffer.size":    "131072"
}
```

Kafka Engine → Materialized View → ReplacingMergeTree

Three CREATE statements per table — that's the whole pattern.

1. `<table>_kafka` — Kafka engine table, reads from Redpanda topic, everything `Nullable`
2. `analytics_<table>` — ReplacingMergeTree, the queryable destination, owns `_version` + `_deleted`
3. `<table>_mv` — Materialized view that types, coalesces, and inserts into (2)

The MV is also the **PII firewall** — we'll come back to that shortly.

Kafka Engine → Materialized View → ReplacingMergeTree

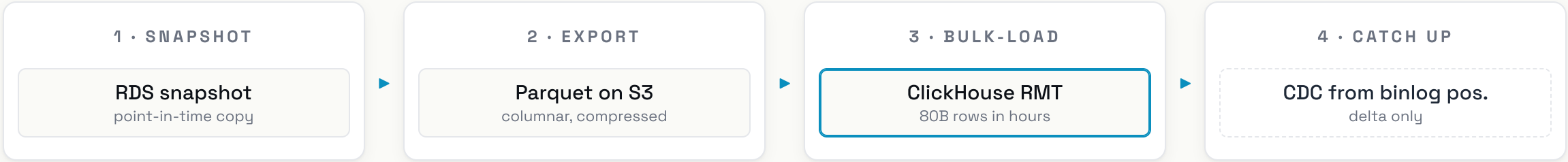
— the three CREATE statements

```
CREATE TABLE orders_kafka (
  id UInt64, workspace_id UInt64, total_amount Nullable(Float64),
  created_at Int64, updated_at Int64, __deleted Nullable(String)
) ENGINE = Kafka SETTINGS
  kafka_broker_list = '${REDPANDA_BROKERS}',
  kafka_topic_list  = 'datalake.app.orders', kafka_format = 'JSONEachRow';
-- ──────────────────────────────────────────────────────────────
CREATE TABLE analytics_orders (
  id UInt64, workspace_id UInt64, total_amount Float64,
  created_at DateTime64(3), updated_at DateTime64(3),
  _version UInt64, _deleted UInt8 DEFAULT 0
) ENGINE = ReplacingMergeTree(_version)
PARTITION BY toYYYYMM(created_at) ORDER BY (workspace_id, id);
-- ──────────────────────────────────────────────────────────────
CREATE MATERIALIZED VIEW orders_mv TO analytics_orders AS SELECT
  id, workspace_id,
  coalesce(total_amount, 0)            AS total_amount,
  fromUnixTimestamp64Milli(created_at) AS created_at,
  fromUnixTimestamp64Milli(updated_at) AS updated_at,
  intDiv(updated_at, 1000)             AS _version,
  if(__deleted = 'true', 1, 0)         AS _deleted
FROM orders_kafka;
```

Don’t Bootstrap 9B Rows Through Debezium

WAR STORY

CDC is for the stream, **not** for hauling history.

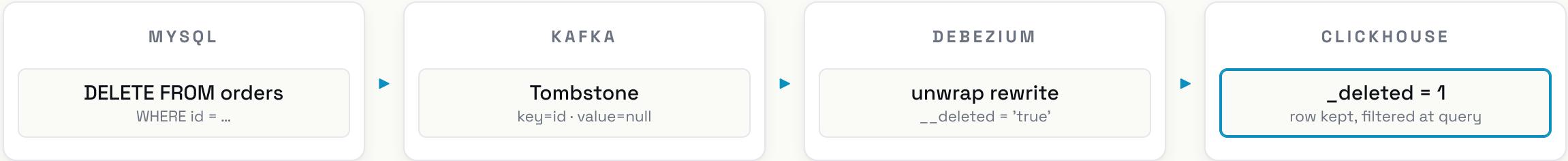


- Snapshotting history through CDC = **days** of replica load + connector risk
- Decoupling cold-start from the live pipeline → primary never feels it
- Then start CDC at the binlog position for the delta — best of both worlds

A Delete Is Just Another Event

WAR STORY

Naïve consumers either drop real deletes or choke on tombstones (null-value records).




```
"transforms.unwrap.delete.handling.mode": "rewrite",
"transforms.unwrap.drop.tombstones":      "true"
```

History kept, churned rows still queryable.

Two Ways Dedup Silently Fails

WAR STORY

RMT dedups **within a partition, by sort key**. Violate either and it quietly keeps duplicates.

- **Mutable column in** `ORDER BY` — a value that changes post-insert produces two rows with different sort keys; RMT keeps both
- **Cross-partition duplicates** — RMT never dedups across partitions; one row reinserted into a new month survives

We caught it by row-count drift: **12.68% extra rows** on one table. Full reload required.

Rules: only immutable columns in `ORDER BY` · never partition on anything CDC can mutate.

Two Ways Dedup Silently Fails

WAR STORY

— the rebuild migration (atomic swap)

```
-- Before: ORDER BY uses a column that can change post-insert → BOTH versions kept
ENGINE = ReplacingMergeTree(_version)
PARTITION BY toYYYYMM(created_at)
ORDER BY (site_id, id)              -- site_id was being reassigned

-- Fix: rebuild with immutable-only sort key
CREATE TABLE analytics_courses_new (...)
ENGINE = ReplacingMergeTree(_version)
PARTITION BY toYYYYMM(created_at)
ORDER BY (id);                      -- id never changes

INSERT INTO analytics_courses_new SELECT * FROM analytics_courses;
OPTIMIZE TABLE analytics_courses_new FINAL;
EXCHANGE TABLES analytics_courses AND analytics_courses_new;
```

The Connector That Lied About Being Fine

WAR STORY

One large MySQL transaction overflowed `binlog.buffer.size`. The connector dutifully reported **RUNNING**. Offsets froze for 6+ hours.

- Shared binlog stream → one stuck transaction stalled **every** workspace connector
- Health is **data moving**, not an API status field
- Compare connector binlog position vs `SHOW MASTER STATUS` — that's the truth
- Fix: buffer 16 KB → 128 KB, plus monitor offsets per partition

The Connector That Lied About Being Fine

WAR STORY

— what RUNNING really meant, and the fix

```
GET /connectors/orders-connector/status
{
  "connector": { "state": "RUNNING" },
  "tasks":     [{ "state": "RUNNING", "trace": null }]
}

// debezium.offsets topic: same binlog position for 6+ hours
```

```
binlog.buffer.size = 131072   ; 16 KB default → 128 KB
max.batch.size     = 2048
max.queue.size     = 8192
```

When NULL Isn't False

WAR STORY

Rails booleans have three states: `0 (false)` · `1 (true)` · `NULL (legacy/unset)`.

- We assumed `NULL = not anonymous` → backfilled **+620M extra rows**
- Worse: `anonymous` is a **generated column** lazily synced from email/phone, so even `= 0` lied for newly-inserted rows
- Fix: filter on the **source identity fields**, not the derived flag

Bonus war story: `mysql()` federation doesn't push down `ORDER BY` / `LIMIT` — treat it as a full scan and chunk by `id` range yourself.

When NULL Isn't False

— the buggy query and the fix

WAR STORY

```
-- The bug: pulled in every NULL-anonymous legacy contact
WHERE anonymous = 0

-- Worse: `anonymous` is a generated column, lazily synced from email/phone.
-- Even `= 0` lied for newly-inserted rows.

-- Fix: filter on the source identity fields, not the derived flag
SELECT id, workspace_id, ...
FROM mysql(app_production, table='contacts')
WHERE (email_address IS NOT NULL AND email_address ≠ '')
   OR (phone_number  IS NOT NULL AND phone_number  ≠ '');
```

PII Never Reaches the Analytics Tables

Two hard boundaries, both enforced **before** any analyst sees a row:

- **At the connector** — Debezium's `column.exclude.list` drops PII columns from the binlog stream. Sensitive bytes never enter the topic.
- **At the materialized view** — explicit column list, no `SELECT *`. PII columns can't accidentally land downstream even if the connector misses one.

Contacts are identified by **presence** of email/phone, never by the values. Analysts get rich behavior; sensitive fields physically don't exist downstream.

PII Never Reaches the Analytics Tables

— the two boundaries

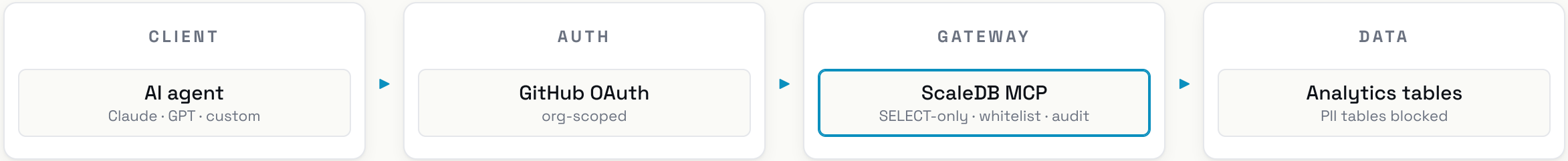
1 · At the connector

```
"column.exclude.list":
  "app.orders.shipping_address_first_name,
   app.orders.shipping_address_last_name,
   app.orders.shipping_address_phone_number,
   app.orders.billing_address_street_one,
   app.orders.billing_address_street_two,
   app.orders.phone_number,
   app.orders.notes,
   app.orders.encryption_key"
```

2 · In the materialized view

```
-- Contacts MV: PII → presence flags only
SELECT id, workspace_id,
  if(email_address ≠ '', 1, 0) AS has_email,
  if(phone_number  ≠ '', 1, 0) AS has_phone,
  if(first_name    ≠ '', 1, 0) AS has_first_name,
  ...
FROM contacts_kafka;
```

Letting AI Agents Query the Lake — Safely



- The access layer is the **control plane** — assume the agent is curious and untrusted
- SELECT-only · whitelisted tables · every query audited
- **Blocks PII tables outright** (users, contacts, memberships)
- Agents get analytics power; they **cannot** read sensitive data — even by accident

Proving the Lake Matches the Source

Drift is inevitable at billions of rows. The trick: find **where** to look before comparing rows.

- **CRC scan** all tables → list mismatched ID batches in minutes
- **Deep scan** only the bad batches → row-by-row column compare
- **Fix mode**: `REPLACE INTO` target from source; re-verify
- **Time fence** all queries to before script-start, so in-flight CDC lag doesn't cause false positives

Cheap fingerprint, expensive only where it matters.

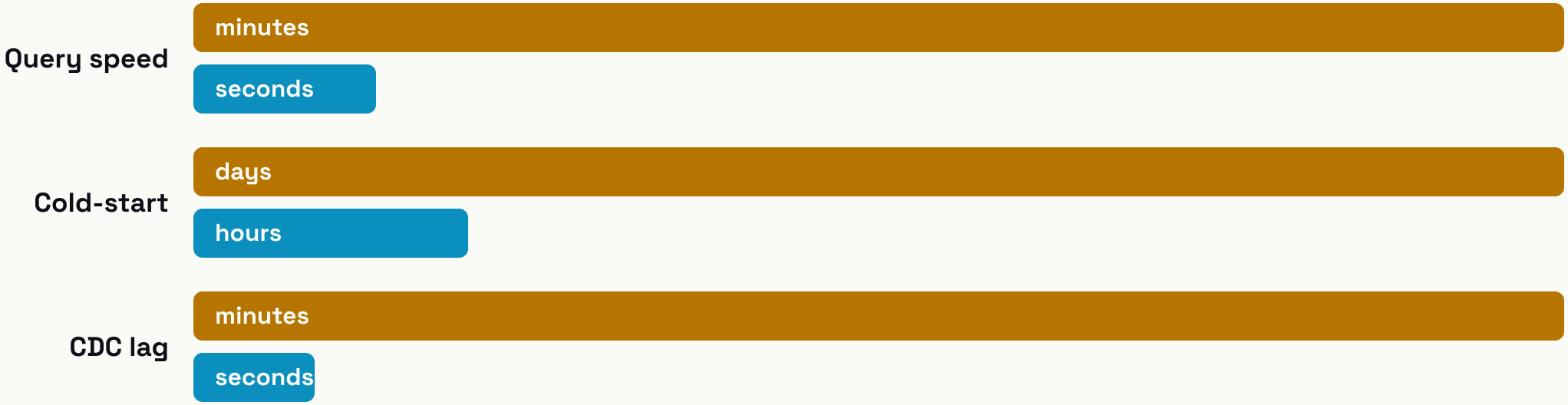
Proving the Lake Matches the Source

— the per-batch fingerprint

```sql
-- Run against source MySQL and target ClickHouse in parallel
SELECT
  COUNT(*),
  COALESCE(SUM(id), 0)                            AS id_sum,
  COALESCE(SUM(UNIX_TIMESTAMP(created_at)), 0)  AS created_sum,
  COALESCE(SUM(UNIX_TIMESTAMP(updated_at)), 0)  AS updated_sum
FROM orders
WHERE id BETWEEN :batch_min AND :batch_max
  AND updated_at < :fence_time;   -- ignore in-flight CDC lag
```

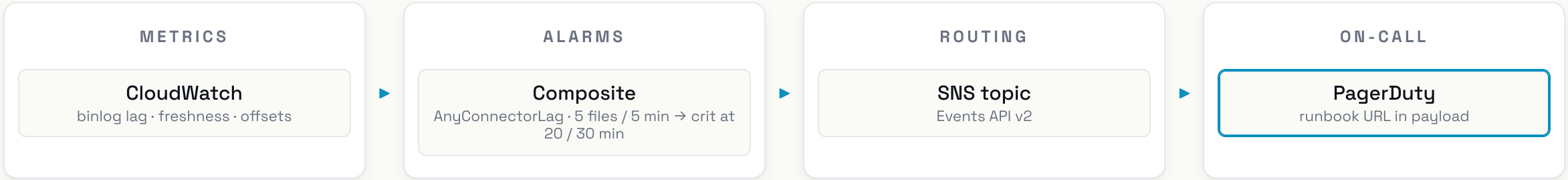
Four cheap aggregates per batch. If any disagree → schedule deep scan. The time fence is the trick: both sides see the same snapshot regardless of replication lag.

What We Got



All of it on **~$4k/month** — for a full real-time lake of 80B+ events.

Things Go Wrong. Plan for It.



- Every node pushes pipeline metrics to CloudWatch — binlog lag, freshness, offsets
- Per-connector + a composite ”any connector behind” alarm
- Runbook URL travels **with** the alert — on-call gets context, not just a red number

Thanks to **PagerDuty** — our sponsor today, and the layer that wakes us up when this thing actually breaks.

If You Build One of These

- Use **CDC for the stream**, but **snapshots for history**
- Never use mutable columns in your **ORDER BY**
- Monitor connector **offsets**, not just API status
- Treat the Materialized View as a **hard PII firewall**
- Verify integrity with **checksums**, not just row counts

> ***The hard part isn't moving data. It's operating the pipeline safely at scale.***

SCALE DB

PIONEERING PERFORMANCE

Questions?

Javier Zon · Founder, ScaleDB · `support@scaledb.io`

LEARN MORE ABOUT US

scaledb.io

CLONE & RUN LOCALLY · DOCKER COMPOSE UP

github.com/scaledb-io/cloud

SCAN FOR THE REPO

PERCONA Live 26 BAY AREA