The Invisible Tax

How Data Format Conversions Drive Up Telemetry Pipeline Costs

Cijo Thomas · Microsoft

KubeCon + CloudNativeCon India 2026

Speaker

Cijo Thomas · Microsoft

Works across **multiple areas of OpenTelemetry** – language implementations (Rust, .NET), Specification, and OTel Arrow.

On the team behind Microsoft's **planetary-scale telemetry pipelines**.

Why Talk About Pipelines?

Telemetry volume keeps growing. The industry has fought back hard on **storage** (tiered, columnar) and on **what gets ingested** (sampling, filtering, drop rules).

The pipeline itself — the path between the app and the backend — gets far less attention.

That's what this talk is about.

What Pipelines Are For

> *Anything between the moment a telemetry signal is produced and the moment it lands in a queryable state in some backend — that's the pipeline.*

Pipelines do real, useful work between the app and the backend:

- **Transport**
- **Batch & compress**
- **Filter & sample**
- **Enrich**
- **Redact**
- **Route**
- **Transform**

Pipelines Cost CPU. A Lot of It.

Every one of those operations runs on real CPU — in collectors, in sidecars, in regional aggregators, on every host. (Memory and network too — but CPU is usually the loudest.)

At scale, **pipeline CPU adds up fast** — running collectors at every hop, on every host, becomes a real line item in the observability bill.

So it's worth asking: **where does that CPU actually go?**

How Cheap Is the Simplest Rename Operation?

before

```
LogRecord {
  name: "user.login.attempt",
  severity_num: 9,
  attributes: [
    ("http.status_code", "200"),
    ("user.id", "u-12345"),
    ("exception.type",
     "java.io.IOException"),
  ],
}
```

after

```
LogRecord {
  name: "user.login.attempt",
  severity_num: 9,
  attributes: [
    ("http.status_code", "200"),
    ("user.id", "u-12345"),
    ("exception.kind",  // ← renamed
     "java.io.IOException"),
  ],
}
```

This rename takes **~30 ns** per record. A single CPU core, in a tight loop, can do **~30 million per second**.

So the actual rename cost is **negligible**.

…but anyone who has run a real telemetry pipeline knows a single core gets nowhere close to that.

What Goes On Inside a Collector?

For every batch the pipeline receives:

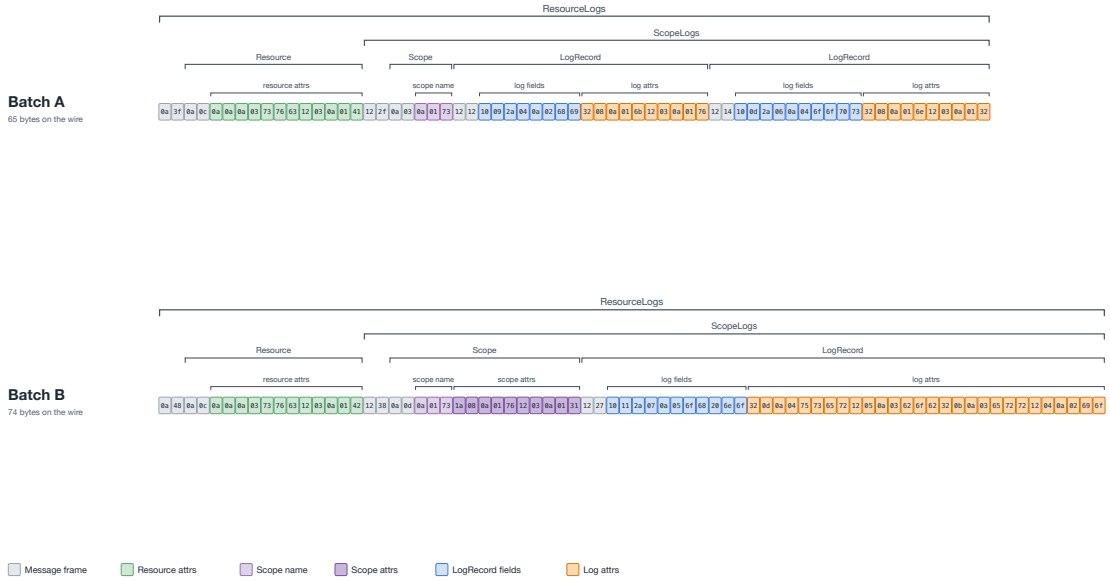
- **Decode** protobuf bytes → allocate a full object graph
- Run any configured processors (filter, transform, route…)
- **Re-encode** back to bytes — and throw the whole object graph away

Even with **zero processors** — pure passthrough — decode + re-encode + GC churn still happens on every batch, every hop.

Add a rename rule: `exception.type → exception.kind`. Throughput drops further. CPU climbs.

Conversion happens **once per batch** — so extra CPU per rule isn't conversion. The **in-memory shape itself** is expensive to walk.

What the Bytes Look Like



Opaque protobuf — every hop has to decode this before it can do anything.

Your Application Pays This Too

Anything emitting telemetry — through any OTel SDK — runs the same shape conversion before bytes leave the process:

```
SDK record → protobuf struct → byte[]
```

Two conversions per record, on every host emitting telemetry.

Measured cost per log record:

SDK	Conv 1 (SDK → struct)	Conv 2 (struct → bytes)	% in Conv 1	**Total**
Rust	395 ns	114 ns	**~77%**	**~510 ns**
Go	281 ns	598 ns	**~32%**	**~887 ns**
.NET (skips Conv 1)	–	–	**0%**	**~194 ns**

.NET emits the protobuf struct directly from the SDK — proof that Conv 1 is avoidable, not free.

This Is a Structural Problem

The Collector and the OTel SDK pay the same three costs — **conversion at every hop**, **GC churn from per-batch allocations**, and **per-row processing**.

The root cause: **every component picks its own in-memory shape**. So conversion happens **at every boundary** — not by design, but by accident.

The wire format (OTLP protobuf) is *only* a wire format. It's nobody's in-memory home.

What Would the Fix Look Like?

Wire transport is unavoidable — bytes have to cross processes and machines. So we need an **in-memory representation** that:

- **Maps cheaply to/from the wire** — minimal decode and encode work at boundaries
- **Fits how pipelines actually work** — bulk per-column ops over batches, not per-record walks
- **Is language-neutral** — SDK in Rust, processor in Go, backend in Java all share the same shape with no conversion

Good news: OTel is already building on one.

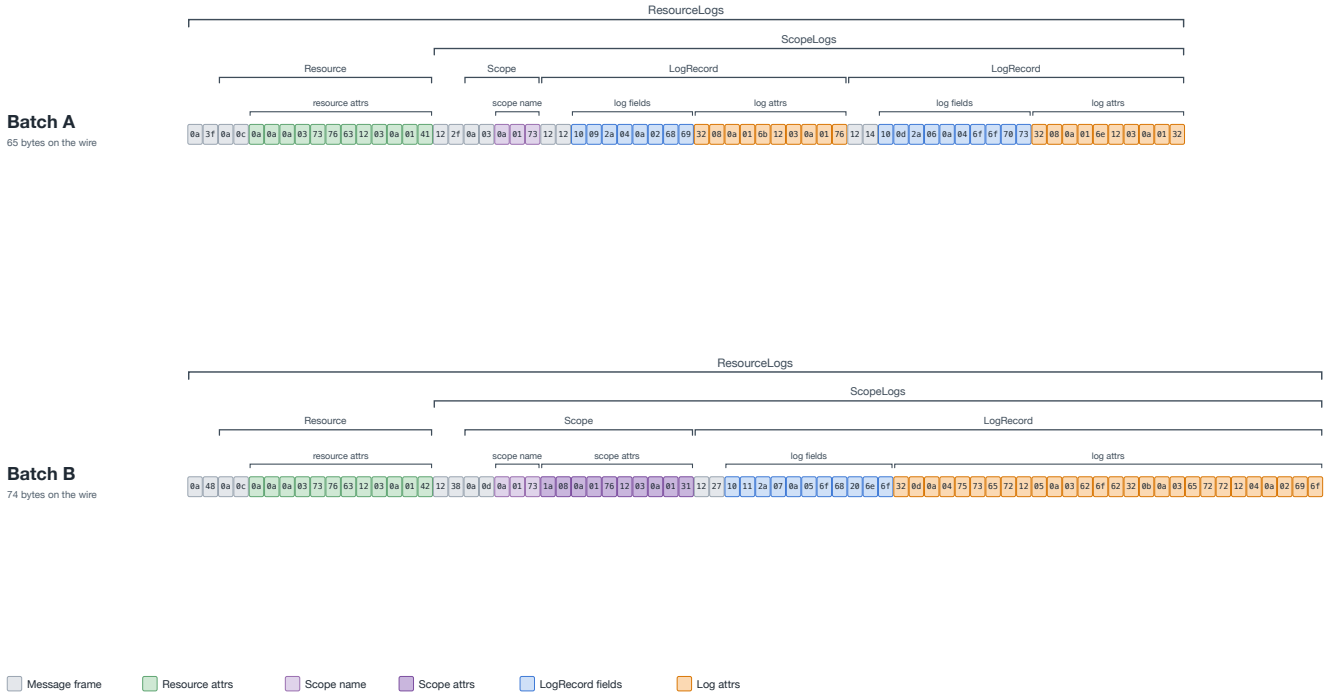
Apache Arrow, OTAP, and the Dataflow Engine

- **Apache Arrow** — a columnar, batch-oriented in-memory format. Same layout on the wire and in memory.
- **OTAP** — *OpenTelemetry Protocol with Apache Arrow*. A 100%-compatible OTLP alternative that carries telemetry as Arrow record batches.
- **Dataflow Engine (DFE)** — a new collector runtime that uses **OTAP as its in-memory representation**, so processors operate directly on Arrow batches.

Together: **no conversion at boundaries. No per-record allocation. Columns, not rows.**

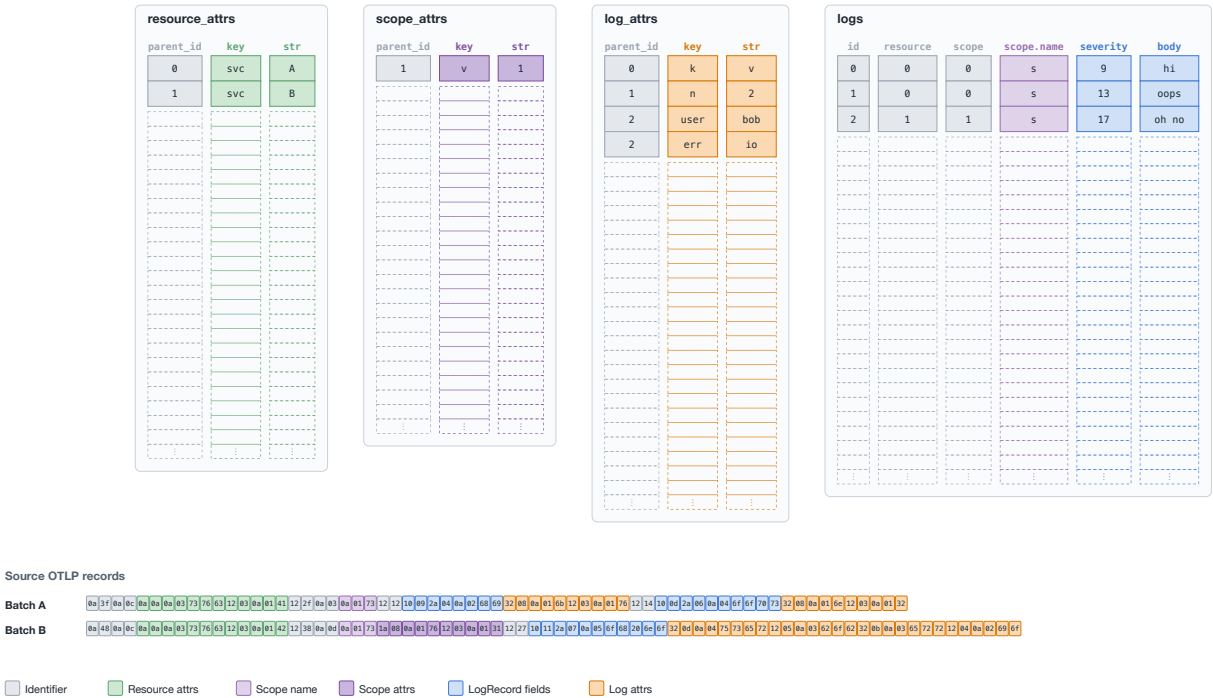
github.com/open-telemetry/otel-arrow

OTLP: OpenTelemetry Protocol



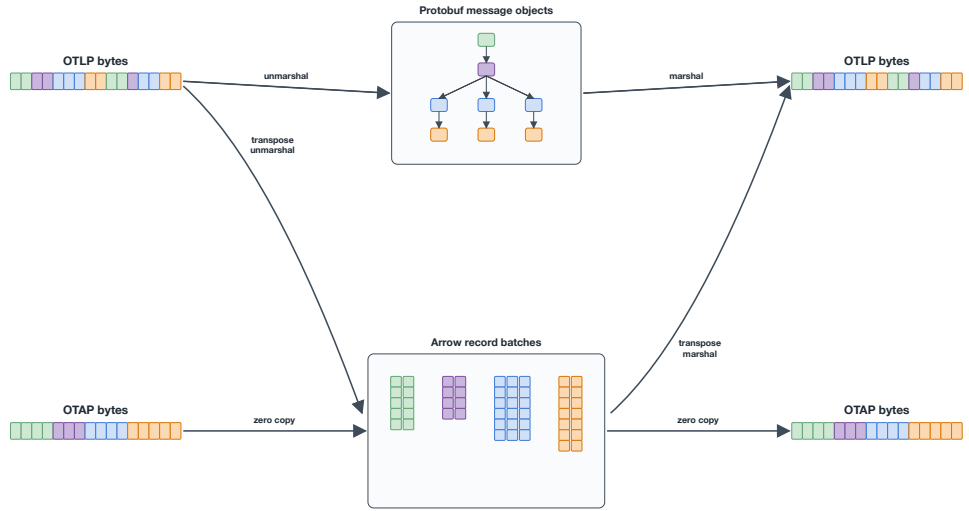
Nested protobuf – opaque until decoded, walked one record at a time.

OTAP: OpenTelemetry Protocol with Apache Arrow



Same data — columnar tables. Wire layout == memory layout.

OTAP In, OTAP Out: No Conversion at All



- Wire bytes **map directly** to Arrow column buffers – no per-record decode
- Processors operate on those batches as **columns**, not as per-record object graphs
- No per-record allocation. No GC churn.

Same Runtime. Only the Protocol Changes.

Both rows are the **same DFE**, on the **same cores**. Only the wire format differs:

Wire protocol	Throughput @ 1 core
OTLP in/out (decode + convert at the boundary)	121K logs/s
OTAP in/out (Arrow end to end)	**2.47M logs/s – ~20×**

Why Arrow: Batch Operations Are Nearly Free

Arrow was designed for **bulk operations over columns**. Renames are exactly that.

Same workload — rename rules added one at a time. **CPU added per rule:**

Rules added	OTel Collector (OTLP)	DFE (OTAP)
+1	**+3.75%**	**+0.07%**
+2	**+7.5%**	**+0.14%**
+3	**+11.25%**	**+0.21%**

OTLP representation walks each record — every rule stacks per-row work. OTAP operates in columns — adding rules is essentially free.

Where Phase 2 Leaves Us

Everything you just saw is the result of **OTel Arrow Phase 2**: OTAP on the wire, the Dataflow Engine, Arrow-native pipeline processing.

Deep dive: OTel-Arrow Phase 2 blog

But we're not done. **Phase 3 starts now.**

Phase 3: Telemetry Born in Arrow

Phase 3 covers a lot of work. A few directions in scope:

- SDKs emit Arrow batches directly — no protobuf step at the app
- Pipelines stay in Arrow end to end — no per-hop conversion
- Backends ingest Arrow — many analytics engines already speak it natively
- **Pluggable components** — run existing collector-contrib components on the new engine

One representation, from source through every hop to query. **Telemetry born in Arrow. Live end-to-end in Arrow.**

Takeaways

- The conversion tax is real, structural, and applies to **every** OTel component — SDKs, collectors, backends.
- The fix is a **shared, columnar, language-neutral in-memory format**. Apache Arrow is the obvious candidate.
- OTel Arrow is exploring exactly this. Early results: **~20× throughput, near-free transforms** on the Arrow-native runtime.
- It's **early** — incubation, not production. The interesting work is just starting.

Like a currency conversion fee: invisible per transaction, painful at scale — and worth removing.

Get Involved

OTel Arrow is **work-in-progress** – early enough that your input shapes it.

- **Telemetry backends**: consider adopting **OTAP as your native ingestion format**. It's early – you can help shape it before the patterns harden.
- **Pipeline operators**: try the **OTAP path / Dataflow Engine** in a non-production setup and share feedback – real-world workloads are exactly what the project needs.
- **Everyone else**: join the community – issues, design discussions, benchmarks all welcome.
- github.com/open-telemetry/otel-arrow
- CNCF Slack #otel-arrow

Thank You & Q&A

Cijo Thomas · @cijothomas · Microsoft

github.com/open-telemetry/otel-arrow
CNCF Slack #otel-arrow