KubeCon CloudNativeCon North America 2026

Two-Stage OTel Aggregation: Taming Istio Cardinality at 10k-Pod Scale

Evelyn Chungyu Chen · Principal Production Engineer, Yahoo Core Infrastructure



Same traffic, 86% fewer series, $140K a year back

ONE PRODUCTION CLUSTER · AT PEAK

800+
nodes

24,000
pods in the mesh

400K
requests / second

▼ Istio metrics — measured off-peak (May 2026), so peak is even higher

2.06M → **280K** active series **−86%**

≈ $140K / year
saved on this one cluster, at list price

What we're going to cover

1. Why Istio emits two million series
2. The OTel data model — two levels, explained with a kindergarten
3. Three processors, and the ordering rule between them
4. Two stages — collapse per node, then across the cluster
5. What broke in production, and what it changed in every query
6. What we saved, and how we proved the numbers
7. One more trap at the vendor, then takeaways



The problem

Why Istio makes so many series — and why the easy fixes don't work

First, what one Istio series actually looks like

```
istio_requests_total{
  reporter="source",
  source_workload="frontend", source_workload_namespace="prod",
  source_app="frontend", source_version="v2", source_cluster="us-east-1",
  source_principal="spiffe://…/frontend",
  destination_workload="checkout", destination_workload_namespace="prod",
  destination_app="checkout", destination_version="v1", destination_cluster="us-east-1",
  destination_service="checkout.prod.svc.cluster.local",
  destination_service_name="checkout", destination_principal="spiffe://…/checkout",
  request_protocol="http", response_code="200", response_flags="-",
  connection_security_policy="mutual_tls", grpc_response_status="",
}  1487
```

…and then the collector adds its own, outside the metric

ADDED BY THE COLLECTOR AND KUBERNETES — OUTSIDE THE METRIC

`k8s.pod.name=frontend-7f9c2-x4k2b` ← new value on every restart

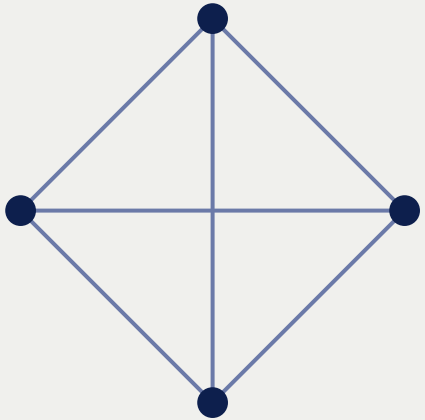
`k8s.node.name=ip-10-4-21-88` · `container.id=3f2a9c1b…`

WHAT ISTIO ITSELF EMITS — INSIDE

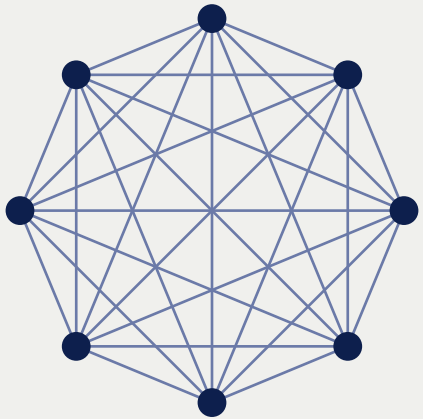
```
istio_requests_total{
  reporter="source",
  source_workload="frontend", source_workload_namespace="prod",
  source_app="frontend", source_version="v2", source_cluster="us-east-1",
  source_principal="spiffe://…/frontend",
  destination_workload="checkout", destination_workload_namespace="prod",
  destination_app="checkout", destination_version="v1", destination_cluster="us-east-1",
  destination_service="checkout.prod.svc.cluster.local",
  destination_service_name="checkout", destination_principal="spiffe://…/checkout",
  request_protocol="http", response_code="200", response_flags="-",
  connection_security_policy="mutual_tls", grpc_response_status="",
}  1487
```

Keep that split in mind — the labels **inside** the metric and the ones added **around** it live on different levels, and that distinction is what the whole pipeline later turns on.

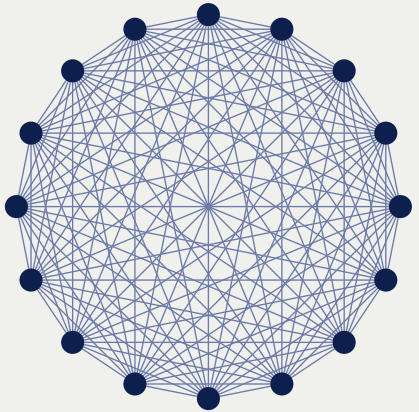
O(N²): at a wedding, everyone greets everyone



4 guests
6 handshakes



8 guests
28 handshakes



16 guests
120 handshakes

Double the guests, and the handshakes roughly quadruple. That's N².

Istio brings the cardinality to next level!

And every pair is then multiplied again

Talking pairs (source × destination)	**N²-ish**
Both proxies report the same call — `reporter=source` and `reporter=destination`	**×2**
× `response_code`, `response_flags`, protocol, …	**×n**
→ `istio_requests_total` — one series per label combination this is a **count**, not an estimate — `count(istio_requests_total)` on our mesh	**29,293**
A histogram repeats **every one of those combinations, once per bucket —** Istio's default duration histogram has 20	**×20**
→ so each histogram is **29,293 × 20 = 585,860**, and there are **3** of them	**1,757,580**
Total, counted: `count({__name__=~"istio_.*"})` 10,000 pods · 374 nodes · off-peak	**2,056,218**

> **The histograms are 85% of the Istio surface** — 1,757,580 of 2,056,218 — which is why "just drop the buckets" is not an answer — those are the latency percentiles everyone actually pages on. The fix has to keep the buckets and still cut the count.

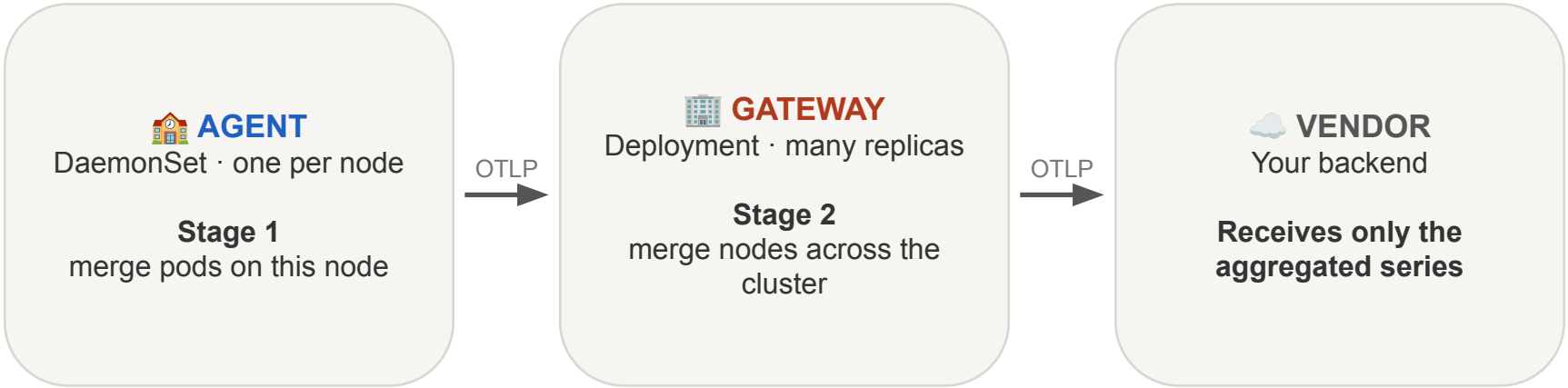
Four options — only one survived

WHAT WE TRIED FIRST

Four options, and what was wrong with each before we landed on OTel processors.

`Vendor aggregation rules`	❌	The aggregation logic lives in the vendor's UI — **switch vendors and you rewrite it**, with no version control and no review
`Prometheus recording rules`	❌	Recording rules aggregate **after ingestion**, so the raw series still land in the TSDB and the bill does not move
`Just send everything`	❌	2.06M series eats the quota outright — **the cost is not acceptable**, and high-cardinality queries do not perform either
`OTel processor aggregation`	✅	**Aggregate before the data ever leaves the cluster.** The config lives in git, gets reviewed, can be rolled back, and is vendor-neutral

Where the aggregation happens: agent → gateway → vendor



Two aggregation stages inside the cluster · only the merged series leave it

KubeCon CloudNativeCon North America 2026

The OpenTelemetry Data Model

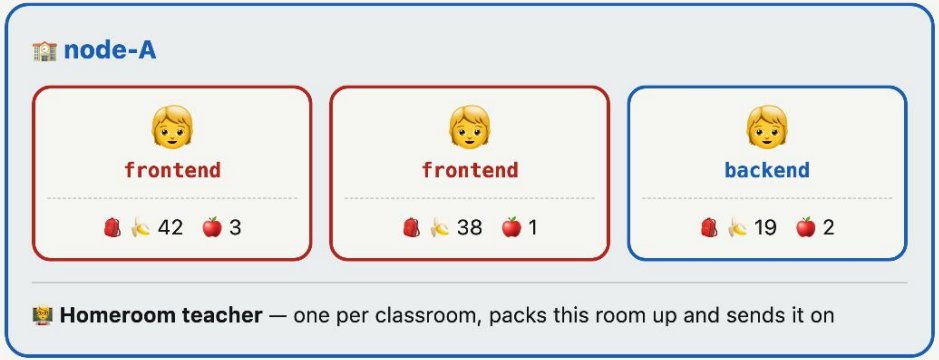
what the data actually looks like



The Kindergarten Mental Model

THE KINDERGARTEN MENTAL MODEL

One picture for the rest of the talk. A classroom is a node, the kids are pods — and what is written **on** the backpack is one layer, what is **inside** it is the other.



Classroom	→	K8s Node
Teacher	→	OTel Agent DaemonSet · one per node
Kids	→	Pods
Backpack	→	Resource attributes k8s.node.name · k8s.pod.name
Fruit inside	→	Datapoint attributes response_code = 503 / 200

> **Outside the bag, inside the bag.** Everything the three processors do is move a label from one of these two places to the other — or sum what is already in the same place.

OpenTelemetry Protocol (OTLP) Specification

The official shape — and where our data sits in it

```
OPENTELEMETRY-PROTO · METRICS.PROTO #L32-L48        →  OUR ISTIO DATA
MetricsData                                         → one export batch
└── ResourceMetrics                                 → one pod's metrics
   ├── 🎒 Resource                                  → k8s.pod.name=frontend-7f9 · k8s.node.name=node-A
   ├── SchemaURL
   └── ScopeMetrics
      ├── Scope                                     → io.opentelemetry.envoy
      ├── SchemaURL
      └── Metric                                    → istio_requests_total
         ├── Name
         ├── Description
         ├── Unit
         └── data
            ├── Gauge
            ├── Sum                                 → istio_requests_total lives here
            │   └── NumberDataPoint                 → one per label combination
            │      ├── 🏷 attributes                → response_code=200 · app=frontend
            │      └── value                        → 1487
            ├── Histogram                           → the three *_bucket families live here
            ├── ExponentialHistogram
            └── Summary
```

🏷️ Resource Level

Top-level metadata (the name tag)

- `k8s.node.name=node-A`
- `k8s.pod.name=frontend-7f9`

Added by K8s & Collector outside the metric payload

🎒 DataPoint Level

Inside the metric payload

- `Metric: istio_requests_total`
- `Attrs: app=frontend, code=200`

Emitted by Istio proxy inside NumberDataPoint

open-telemetry/opentelemetry-proto · opentelemetry/proto/metrics/v1/metrics.proto #L32-L48, plus the NumberDataPoint message from the same file · v1.11.0 — structure verbatim, colour ours

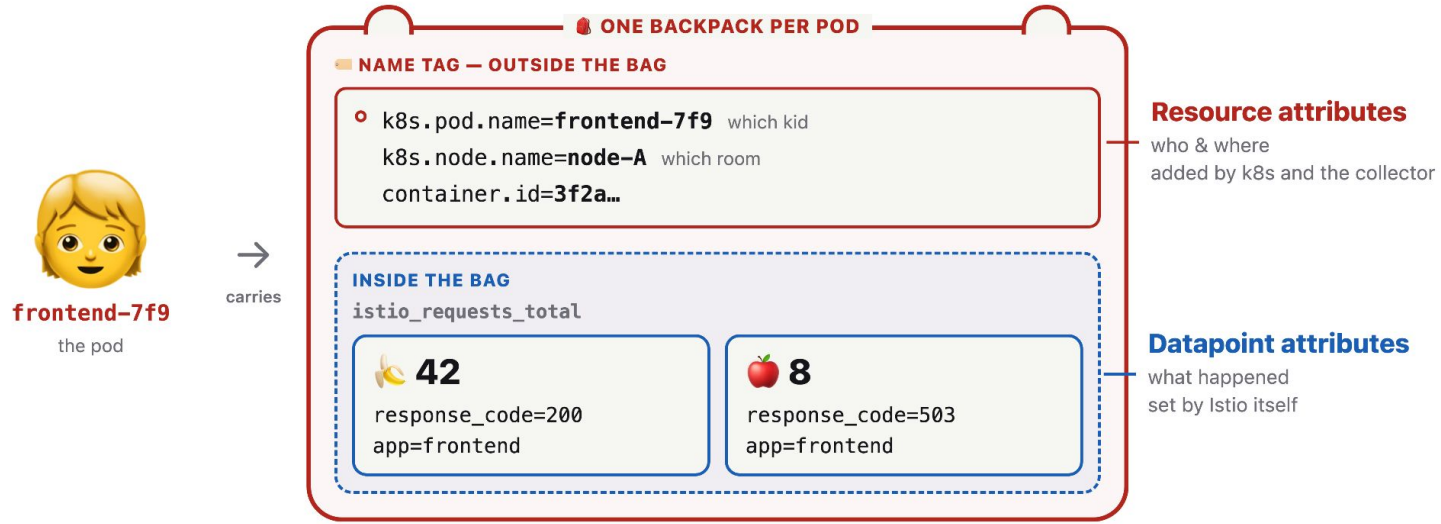
> What Prometheus calls a **Counter**, OTLP calls a **Sum** — and it is **cumulative**. Hold that word.

Outside the backpack vs inside it

WHAT PROMQL SHOWS YOU — ONE FLAT STRING

```
istio_requests_total{app="frontend", response_code="200", k8s_pod_name="frontend-7f9", k8s_node_name="node-A",
container_id="…"}
```

↓ but inside OpenTelemetry, it is two layers



Three processors, one at a time

Three pods on one node: two 🔴 frontends, one 🔵 backend



① Processor - Transform

Stripping per-pod identity attributes at the resource level

otel-collector-config.yaml

```yaml
processors:
  transform/envoy-normalize-resource:
    error_mode: ignore
    metric_statements:
      - context: resource
        statements:
          - delete_key(attributes, "k8s.pod.name")
          - delete_key(attributes, "k8s.pod.uid")
          - delete_key(attributes, "container.id")
```

1. Target Resource Level

Setting **context: resource** explicitly targets top-level ResourceAttributes (the name tag), leaving inner metric datapoints untouched.

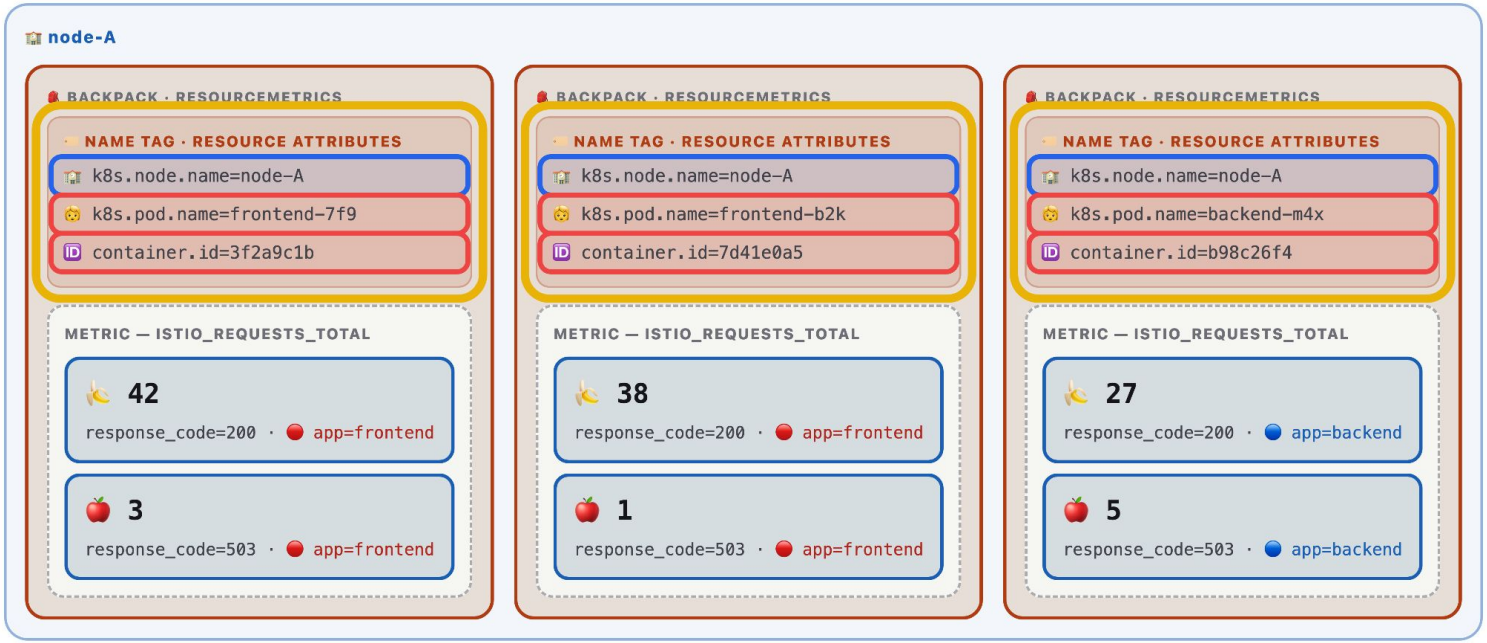
2. Strip Churning Pod Keys

Deletes k8s.pod.name, k8s.pod.uid, and container.id. These attributes change on every restart, causing massive series churn.

3. Preserve Node Context

k8s.node.name is intentionally retained here so the DaemonSet collector can still collapse pods per node in Stage 1.

① Processor - Transform by deleting keys



1. Resource-Level Scope

2. Pod Churn Drives Cardinality

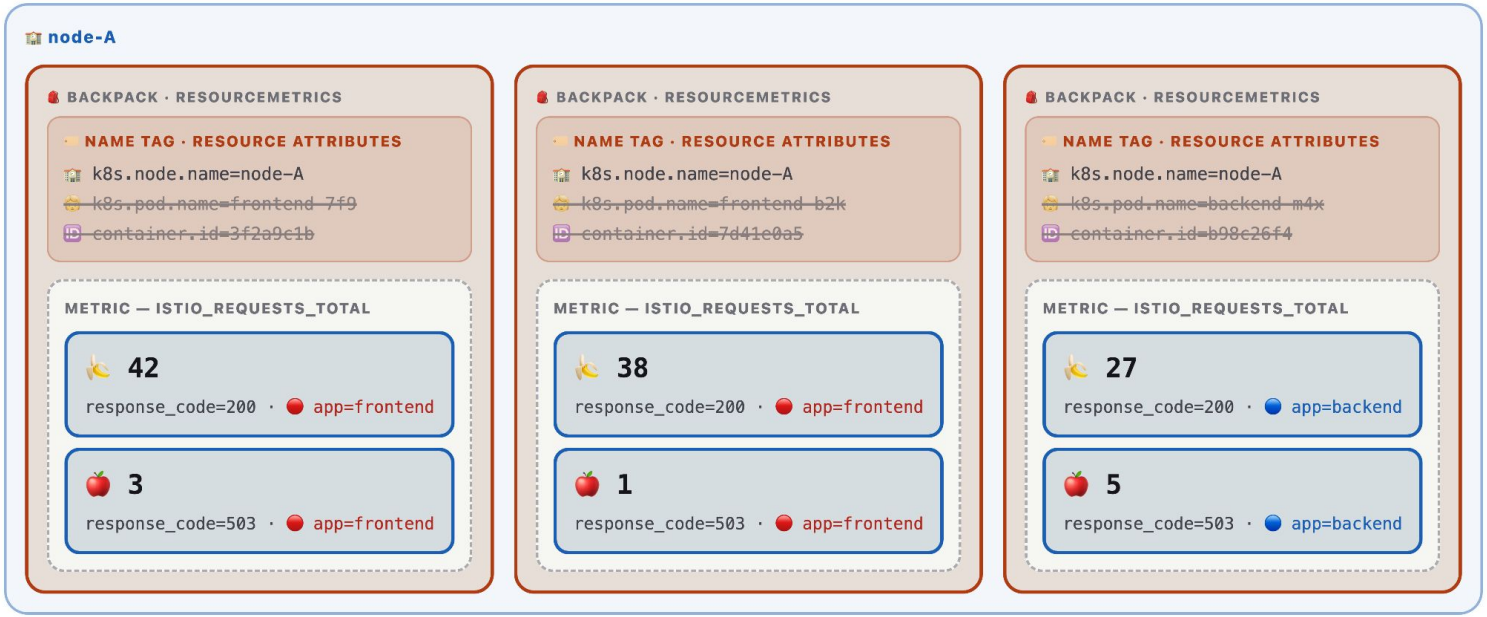
3. Node Context Preserved

① Processor - Transform - Example

Visualizing per-pod attribute stripping on node-A after transformation

Resource level →

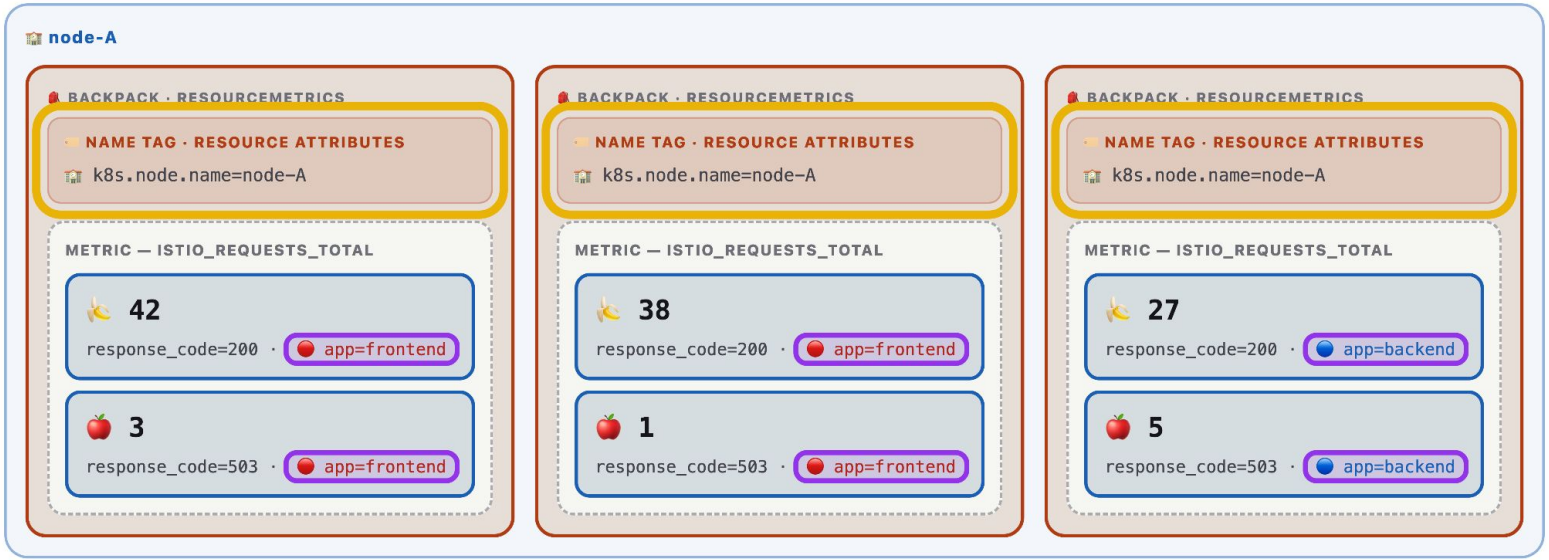
Datapoint level →



② groupbyattrs — app is still tagging the backpack

Resource level →

Datapoint level →



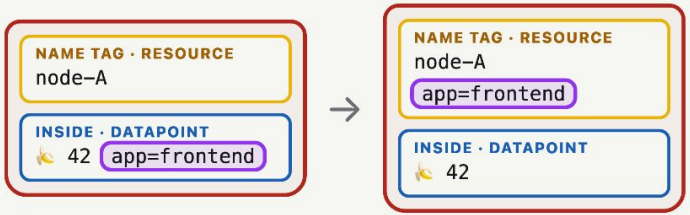
app · what we group by

② Processor — batch and groupbyattrs

```yaml
processors:
  # make the batch big first — groupbyattrs only merges Resources inside one batch
  batch/envoy-pre-groupby: {}

  groupbyattrs/envoy-by-workload:
    keys:
      - app
```

1 **app moves up a layer**



trapped inside the datapoint → promoted onto the name tag

2 **same name tag → same backpack**

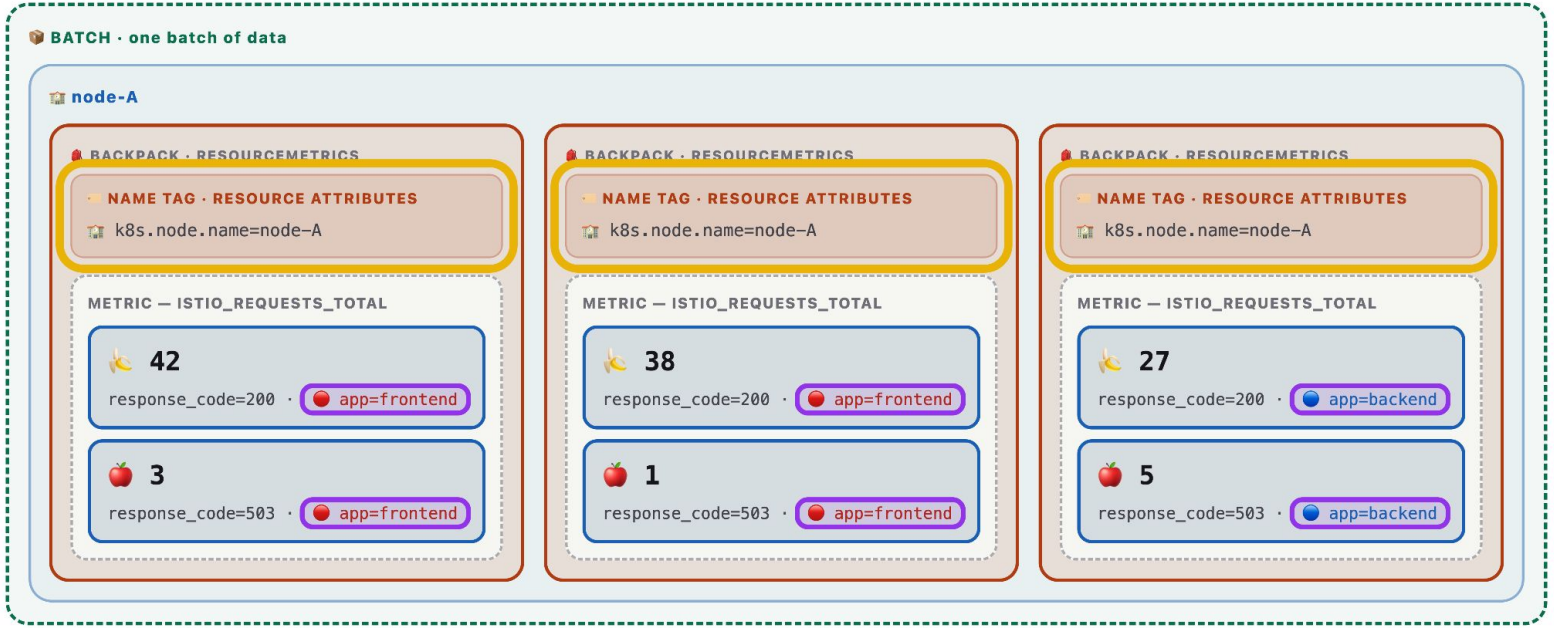


frontend sits together, backend sits together

② Processor - Batch

Resource level →

Datapoint level →

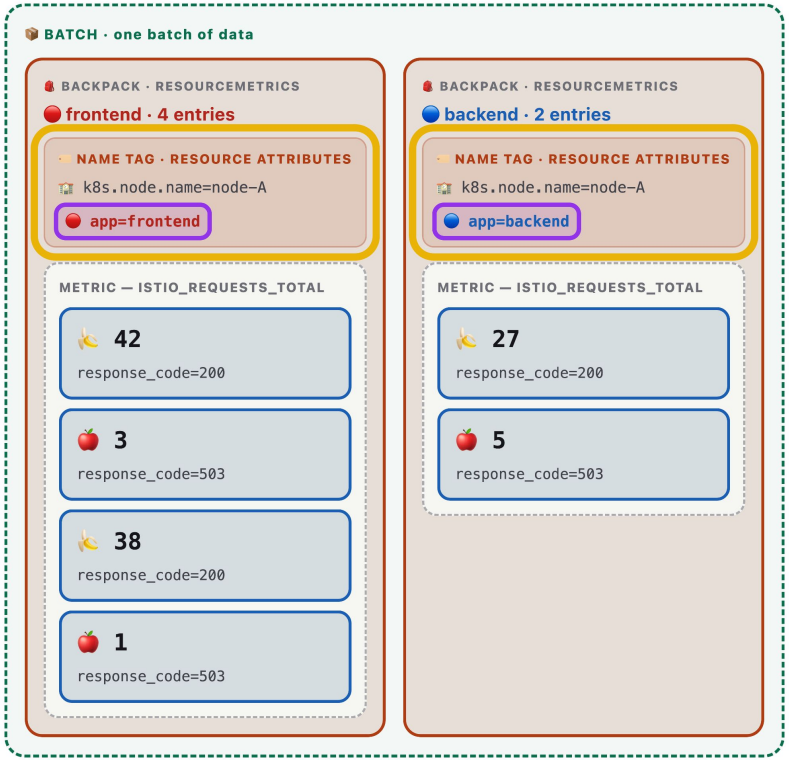


app · what we group by

② Processor - Groupbyattrs

Resource level →

Datapoint level →



app · what we group by

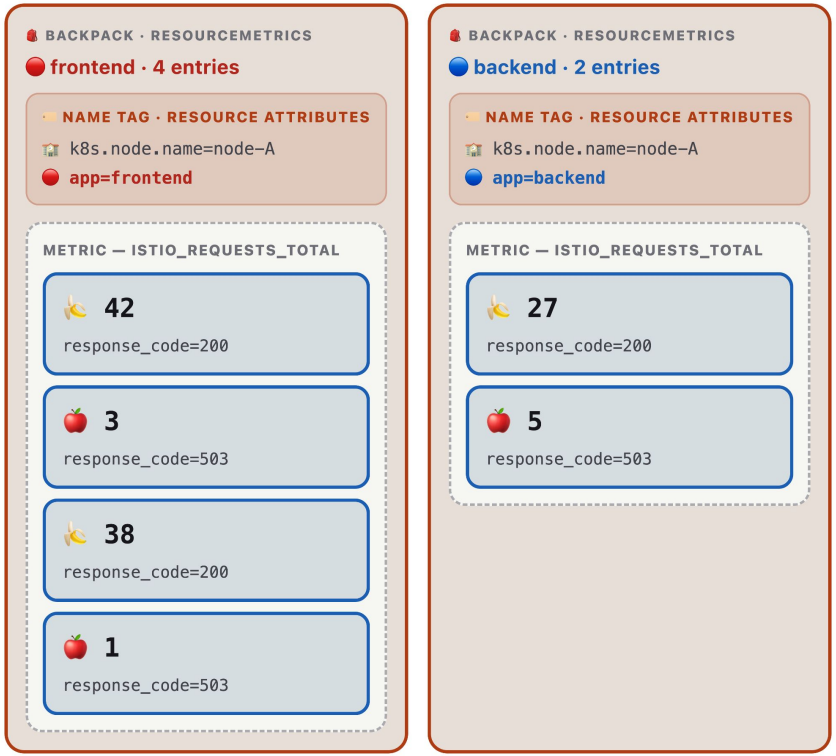
③ Processor - Metricstransform

```
# One action does both halves: keep the whitelist, sum everything else away.
processors:
  metricstransform/envoy-aggregate:
    transforms:
      - include: ^istio_
        match_type: regexp
        action: update
        operations:
          - action: aggregate_labels
            aggregation_type: sum
            label_set:
              - source_workload                  - destination_version
              - source_workload_namespace        - destination_service
              - source_app                       - request_protocol
              - source_version                   - response_code
              - destination_workload             - response_flags
              - destination_workload_namespace   - connection_security_policy
              - destination_app
            # note: app is NOT here — groupbyattrs already moved it
            # up to the resource level, where this whitelist cannot reach it.
```

③ Processor - Before metricstransform

Resource level →

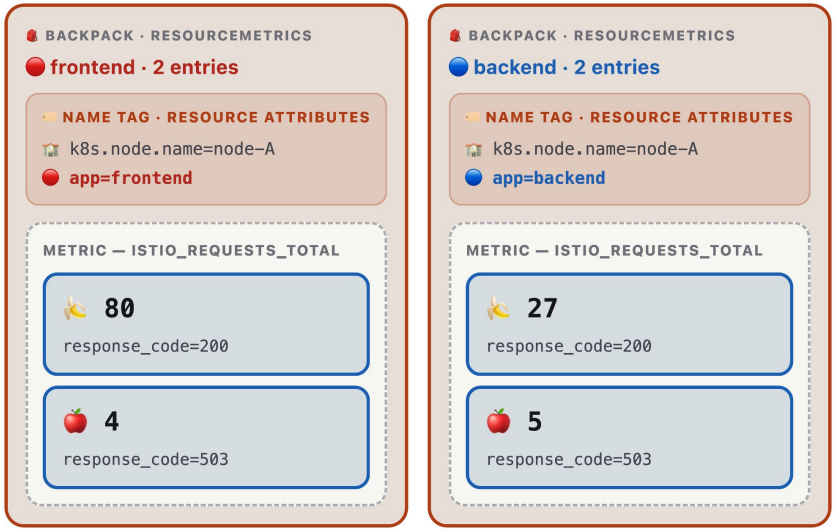
Datapoint level →



③ Processor - After metricstransform

Resource level →

Datapoint level →



Three processors = one line of PromQL

Knowing there are two levels is what lets you read these three

Just the names, and the one setting that matters on each. **We are not explaining them here** — we run them one at a time, on real data, right after this.

processor	the knob	context level	its share of that one query
`transform`	`delete_key`	**resource** the name tag	the part you never write — those labels are gone
`groupbyattrs`	`keys: […]`	**datapoint → resource** promote, then merge	**the by (…)** — decides what is one group
`metricstransform`	`aggregate_labels`	**datapoint** inside the backpack	**the sum** — plus the whitelist of what survives

All three add up to one query you already know

PromQL writes grouping and summing as **one word**. OTel splits them into three processors — and the split is where the ordering bug lives.

```
sum by (app, response_code) (istio_requests_total)
|      |    |
|      |    └──── metricstransform · the whitelist decides what survives
|      └───────── groupbyattrs · promoted app — so this is what "one group" means
└──────────────── metricstransform · the sum itself

transform is not in this query at all. It deleted pod.name and container.id from the identity —
the part you never have to write, because those labels are already gone.
```



Stage 1 · DaemonSet, Stage 2 · Gateway

One pipeline, two passes — and the two places they differ



Two stages, same processors, two differences

DS · one per classroom

2 · TRANSFORM

```
transform:
  delete_key:
    - k8s.pod.name
    - k8s.pod.uid
    - container.id
```

3 · GROUPBYATTRS

```
groupbyattrs:
  keys: [app]
```

4 · METRICSTRANSFORM

```
metricstransform:
  action: aggregate_labels
  label_set: [app, …]
  aggregation_type: sum
```

gRPC / OTLP ▼ one hop, and this room's subtotals leave the machine

GW · front office, many replicas

2 · TRANSFORM — CHANGED

```
transform:
  delete_key:
    - k8s.node.name
    - host.name
```

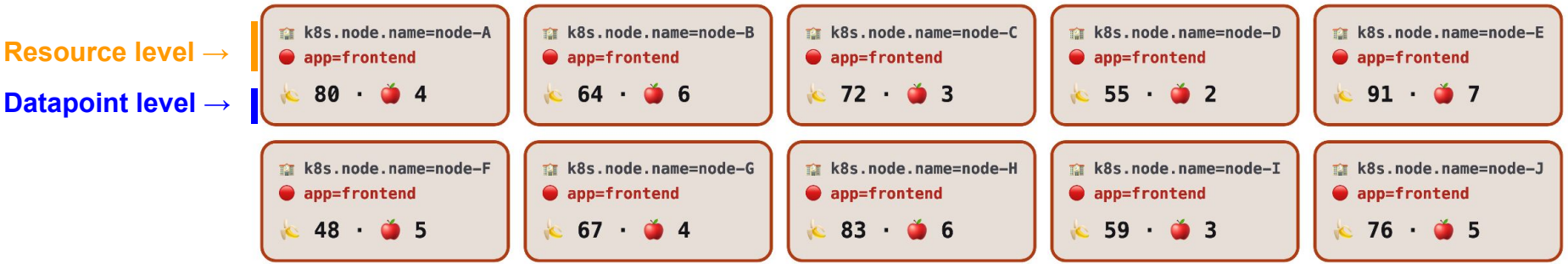
3 · GROUPBYATTRS — CHANGED

```
groupbyattrs:
  keys: []
```

4 · METRICSTRANSFORM — SAME AS DS

```
metricstransform:
  action: aggregate_labels
  label_set: [app, …]
  aggregation_type: sum
```

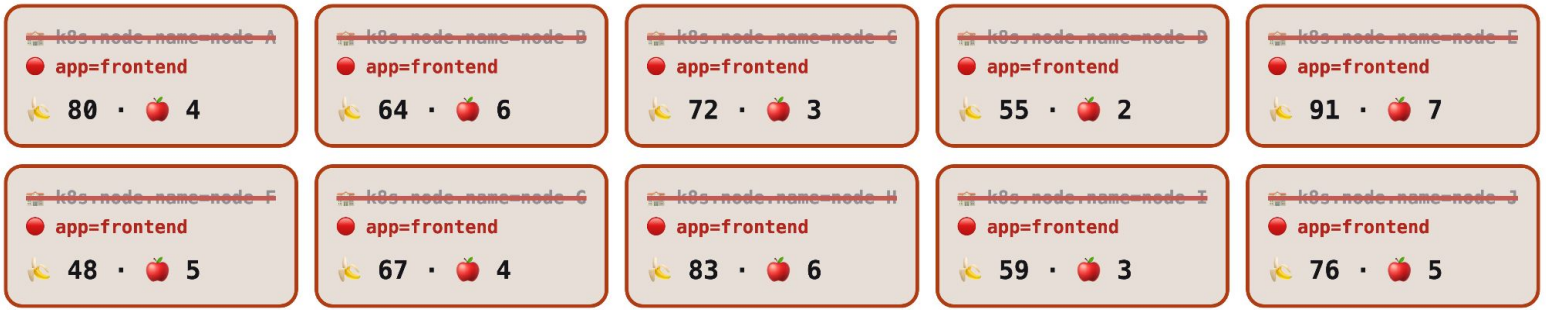
Gateway: ten nodes, one app



Gateway · transform — strip the node name

Resource level →

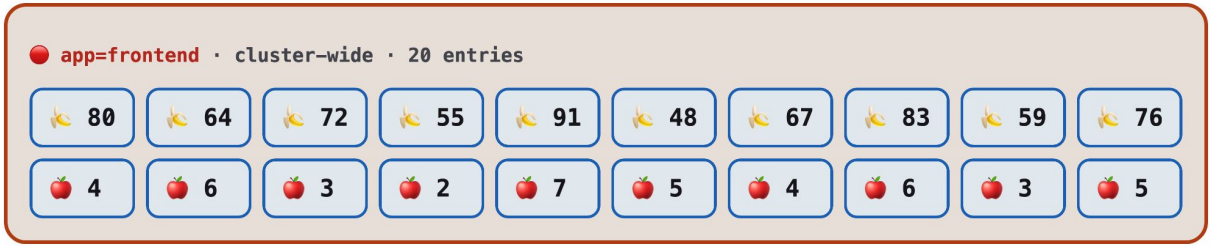
Datapoint level →



Gateway · batch + groupbyattrs — merge only

Resource level →

Datapoint level →



Gateway · metricstransform — sum across nodes

Resource level →

Datapoint level →

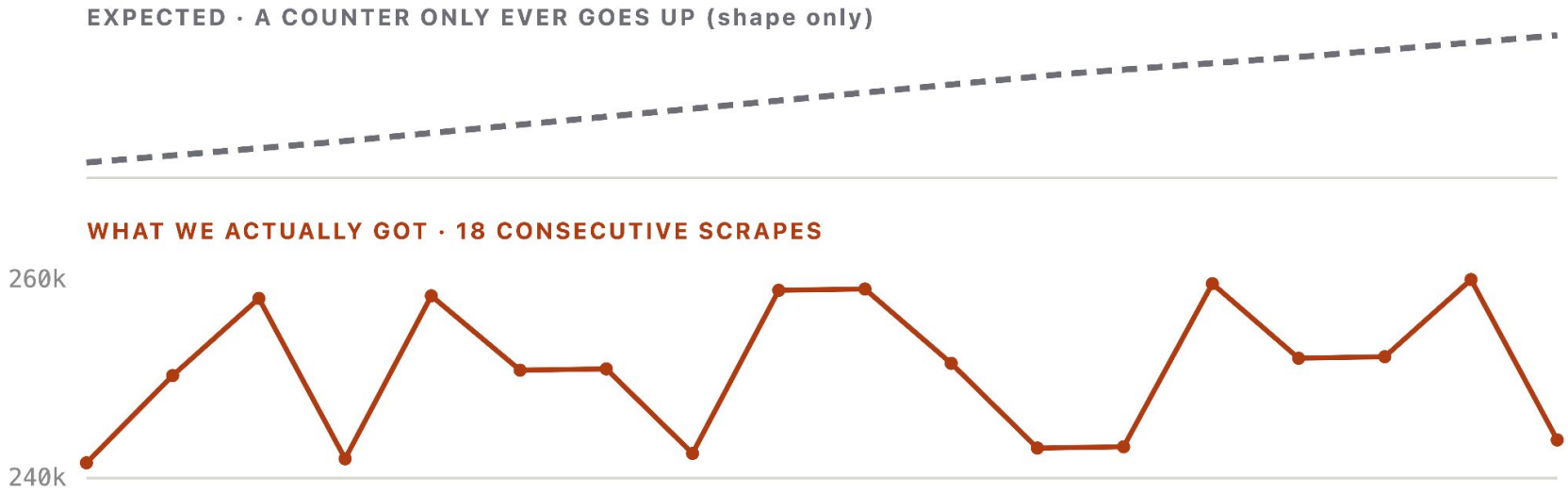


Shipped it to production

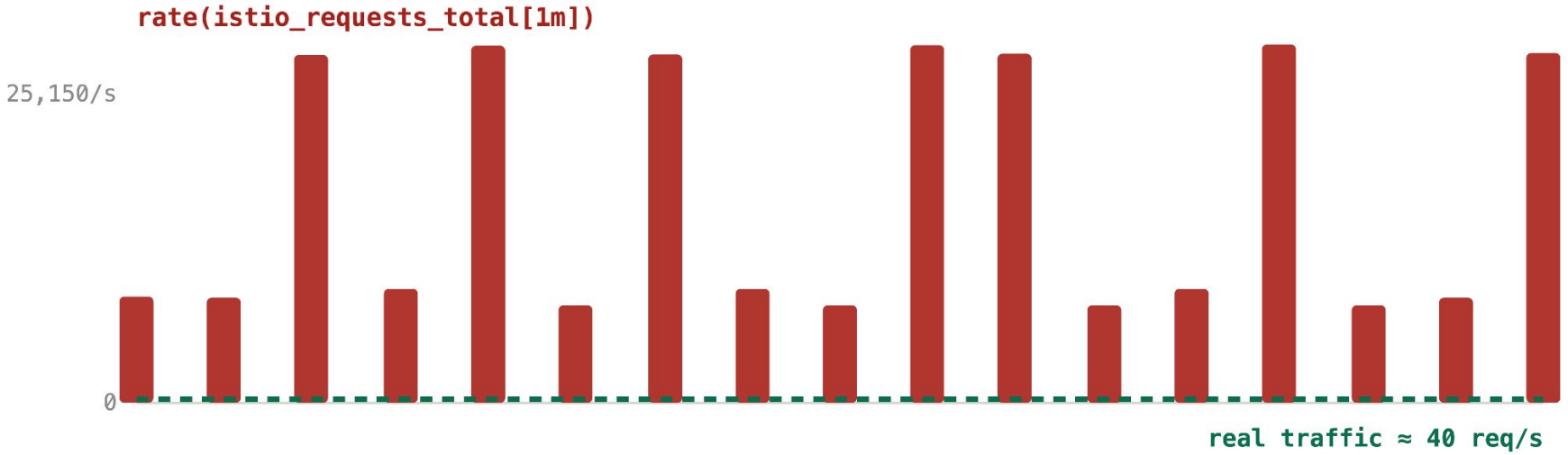


The config was right. The data was wrong.

A counter should only go up — ours didn't



Our RPS dashboard went wild too



rate() inflated 100–635×

We pulled the raw scrapes

t+0s	t+10s	t+20s	t+30s	t+40s	t+50s
241,133	249,766	257,399	241,532 went down	257,665	250,298 went down

Sum two counters, and one restart breaks the total

Cumulative 🍌 counted since the kid arrived at school

	t+0s	t+10s	t+20s	t+30s
👦 pod A	100	110	120	130
👦 pod B	50	60	⟲ 3 restarted, counts from 0	13
sum (pod name stripped)	150	170	123 ↓ went down = a reset to rate()	143

Delta: a restart costs one point, not the whole total

Cumulative 🍌 counted since the kid arrived at school

	t+0s	t+10s	t+20s	t+30s
👴 pod A	100	110	120	130
👴 pod B	50	60	⟲ 3 restarted, counts from 0	13
sum (pod name stripped)	150	170	123 ↓ went down = a reset to rate()	143

Delta 🍌 new in these 10 seconds

at t+20s, true ≈ +20 · cumulative → rate() reads **+123** · delta → **+10**

	t+0s	t+10s	t+20s	t+30s
👴 pod A	–	+10	+10	+10
👴 pod B	–	+10	– restarted, point dropped	+10
sum (pod name stripped)	–	+20	+10 missed pod B's 10s	+20

cumulativetodelta must run first

```
# Must run first: the state key needs full identity, before any resource attr is stri
processors:
  cumulativetodelta:
    initial_value: drop
```

Back to the pipeline — slot 1 filled in

DS · one per classroom

1 · CUMULATIVETODELTA NEW

```
cumulativetodelta:
  include:
    metrics: ["istio_.*"]
    match_type: regexp
```

2 · TRANSFORM

```
transform:
  delete_key:
    - k8s.pod.name
    - k8s.pod.uid
    - container.id
```

3 · GROUPBYATTRS

```
groupbyattrs:
  keys: [app]
```

4 · METRICSTRANSFORM

```
metricstransform:
  action: aggregate_labels
  label_set: [app, …]
  aggregation_type: sum
```

gRPC / OTLP ▼ the data is **already delta** by now

GW · front office, many replicas

2 · TRANSFORM CHANGED

```
transform:
  delete_key:
    - k8s.node.name
    - host.name
```

3 · GROUPBYATTRS CHANGED

```
groupbyattrs:
  keys: []
```

4 · METRICSTRANSFORM SAME AS DS

```
metricstransform:
  action: aggregate_labels
  label_set: [app, …]
  aggregation_type: sum
```

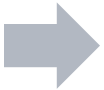
From cumulative counters to deltas

Cumulative counter

`0 → 30 → 60 → 80`

only ever goes up

`rate()` **works ✓**



Delta

`+30 → +30 → +20`

only this interval

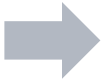
`rate()` **breaks ✗**

Same traffic, new shape: every query built on `rate()` has to change.

rate() breaks silently: every query must change

RPS

```
rate(istio_requests_total[5m])
```



```
sum_over_time(istio_requests_total[5m])
/ 300
```

Error rate

```
rate(5xx[5m])
/ rate(total[5m])
```



```
sum_over_time(5xx[5m])
/ sum_over_time(total[5m])
```

window cancels out: no / 300

Rewriting is quick. Forgetting is silent: re-check every alert.

Results

What it cost and how we proved it



The bigger your mesh, the more you save

BEFORE → AFTER

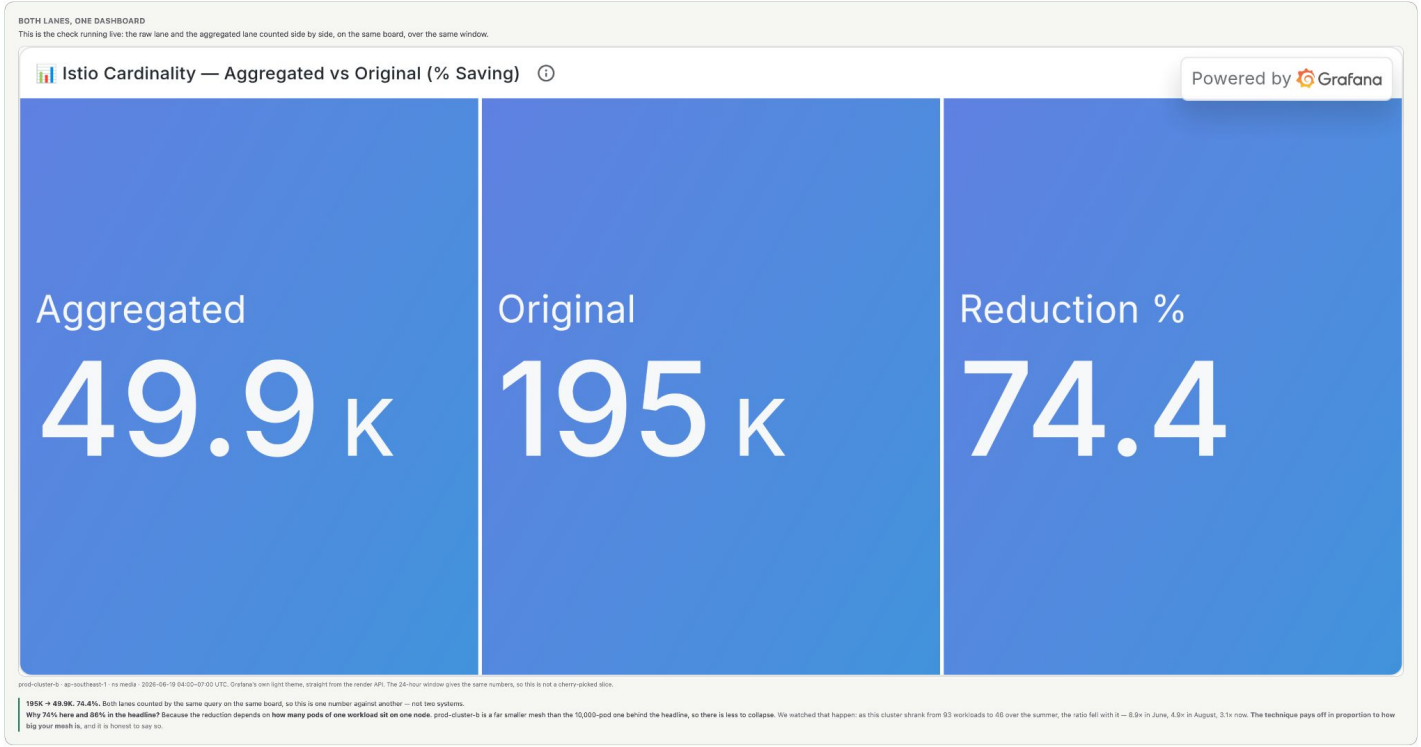
2,060,000 → 280,000

A high-cardinality tidal wave turned into 86% fewer, far more valuable business series — the TSDB bill flattens and the system stops running at the edge.

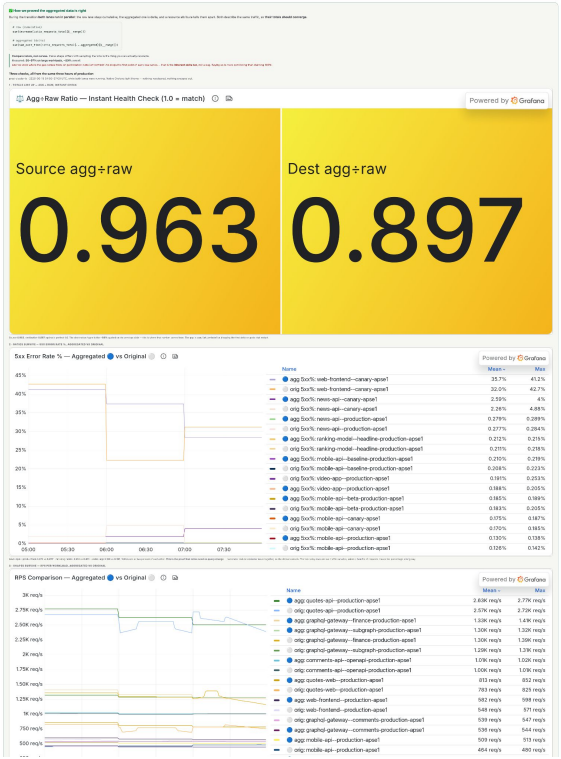
≈ $140,000 / year

at list price, on this one cluster (~$6.50 per 1,000 series per month)

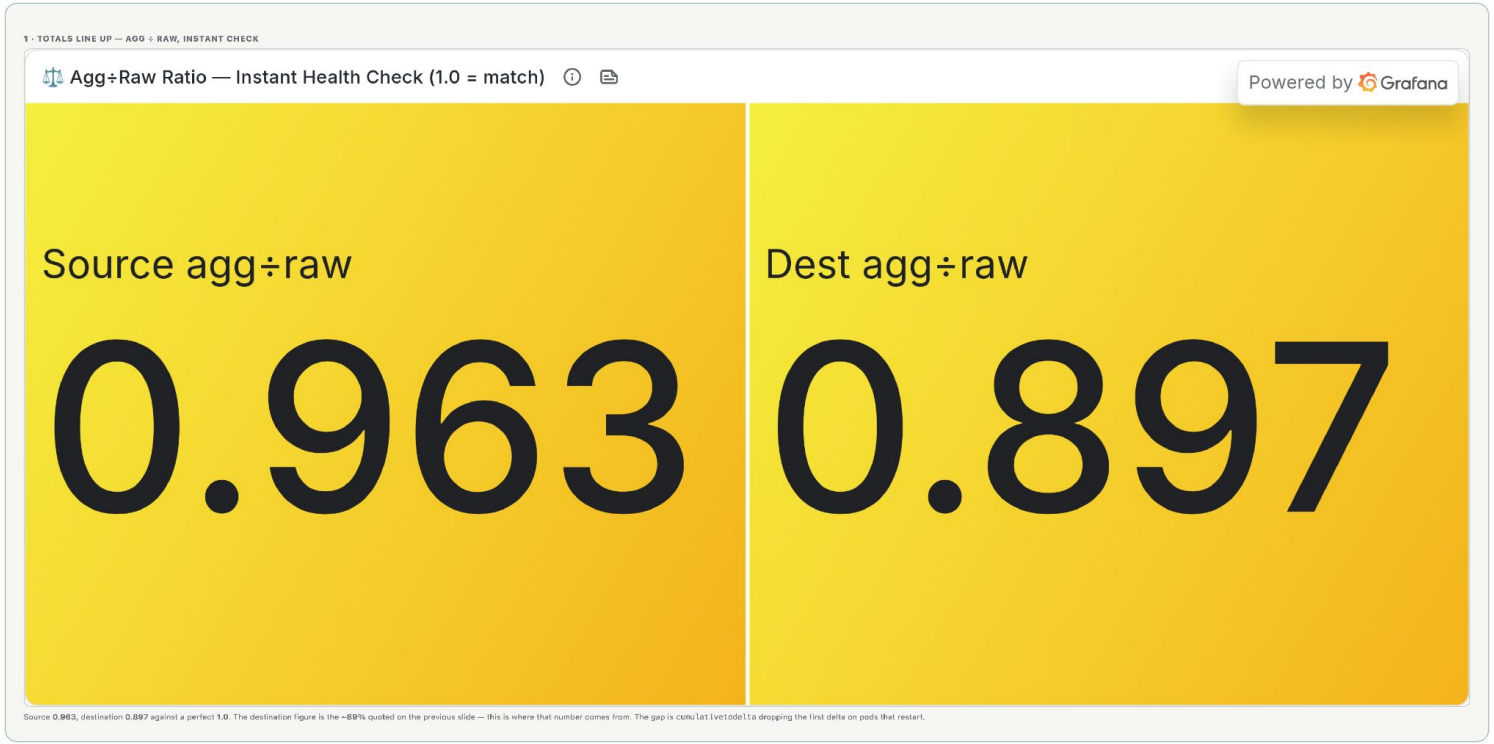
Both lanes, one dashboard — 195K → 49.9K



How we proved the aggregated data is right

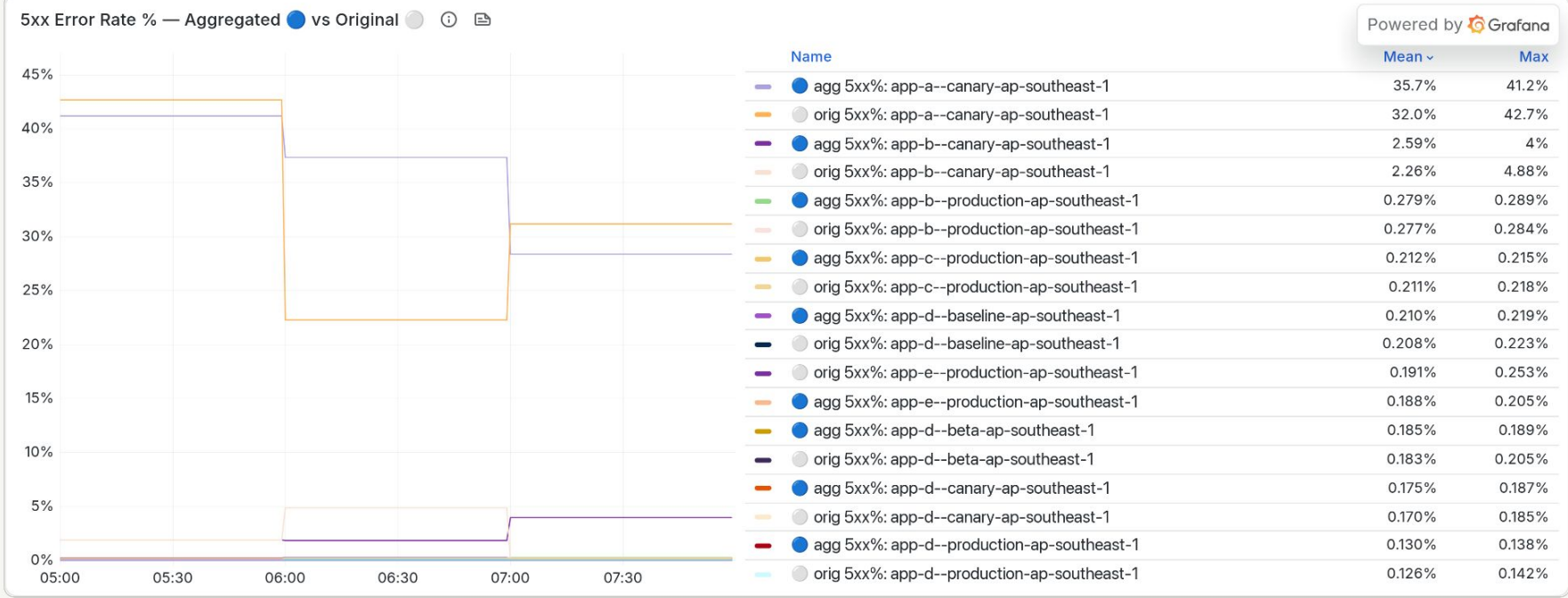


The ratio lines up



5xx errors line up

2 · RATIOS SURVIVE — 5XX ERROR RATE %, AGGREGATED VS ORIGINAL



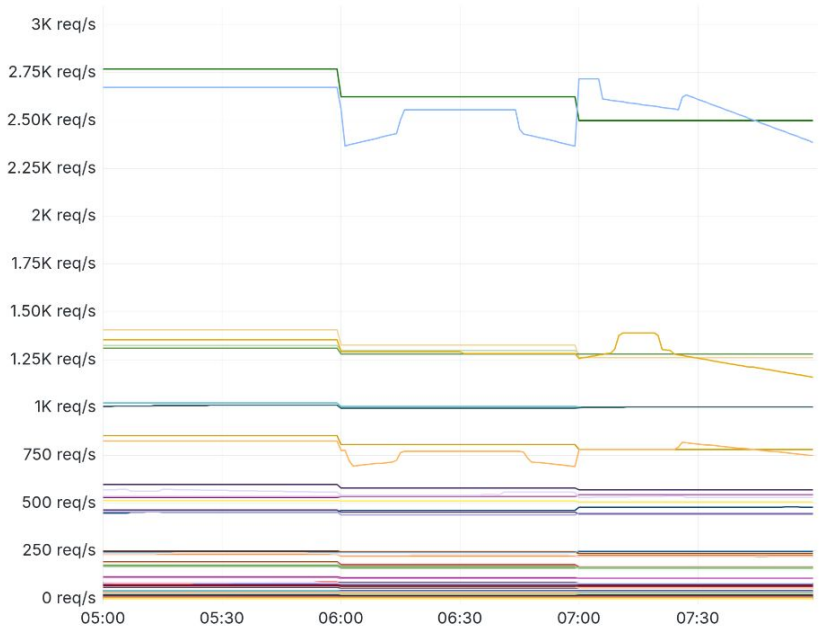
Name	Mean	Max
🔵 agg 5xx%: app-a--canary-ap-southeast-1	35.7%	41.2%
⚪ orig 5xx%: app-a--canary-ap-southeast-1	32.0%	42.7%
🔵 agg 5xx%: app-b--canary-ap-southeast-1	2.59%	4%
⚪ orig 5xx%: app-b--canary-ap-southeast-1	2.26%	4.88%
🔵 agg 5xx%: app-b--production-ap-southeast-1	0.279%	0.289%
⚪ orig 5xx%: app-b--production-ap-southeast-1	0.277%	0.284%
🔵 agg 5xx%: app-c--production-ap-southeast-1	0.212%	0.215%
⚪ orig 5xx%: app-c--production-ap-southeast-1	0.211%	0.218%
🔵 agg 5xx%: app-d--baseline-ap-southeast-1	0.210%	0.219%
⚪ orig 5xx%: app-d--baseline-ap-southeast-1	0.208%	0.223%
⚪ orig 5xx%: app-e--production-ap-southeast-1	0.191%	0.253%
🔵 agg 5xx%: app-e--production-ap-southeast-1	0.188%	0.205%
🔵 agg 5xx%: app-d--beta-ap-southeast-1	0.185%	0.189%
⚪ orig 5xx%: app-d--beta-ap-southeast-1	0.183%	0.205%
🔵 agg 5xx%: app-d--canary-ap-southeast-1	0.175%	0.187%
⚪ orig 5xx%: app-d--canary-ap-southeast-1	0.170%	0.185%
🔵 agg 5xx%: app-d--production-ap-southeast-1	0.130%	0.138%
⚪ orig 5xx%: app-d--production-ap-southeast-1	0.126%	0.142%

`app-b--production` 0.279 vs **0.277** · `app-c` 0.212 vs **0.211** · `app-e` 0.188 vs **0.191**. Within one or two percent of each other. **This is the proof that ratios need no query change** — numerator and denominator move together, so the divisor cancels. The two noisy rows are low-traffic canaries, where a handful of requests moves the percentage a long way.

RPS lines up

3 · SHAPES SURVIVE — RPS PER WORKLOAD, AGGREGATED VS ORIGINAL

RPS Comparison — Aggregated 🔵 vs Original ⚪



Powered by Grafana

Name	Mean	Max
🔵 agg: app-f--production-ap-southeast-1	2.63K req/s	2.77K req/s
⚪ orig: app-f--production-ap-southeast-1	2.57K req/s	2.72K req/s
🔵 agg: app-h--finance-ap-southeast-1	1.33K req/s	1.41K req/s
🔵 agg: app-h--subgraph-ap-southeast-1	1.30K req/s	1.32K req/s
⚪ orig: app-h--finance-ap-southeast-1	1.30K req/s	1.39K req/s
⚪ orig: app-h--subgraph-ap-southeast-1	1.29K req/s	1.31K req/s
🔵 agg: app-i--production-ap-southeast-1	1.01K req/s	1.02K req/s
⚪ orig: app-i--production-ap-southeast-1	1.00K req/s	1.01K req/s
🔵 agg: app-g--production-ap-southeast-1	813 req/s	852 req/s
⚪ orig: app-g--production-ap-southeast-1	783 req/s	825 req/s
🔵 agg: app-a--production-ap-southeast-1	582 req/s	598 req/s
⚪ orig: app-a--production-ap-southeast-1	548 req/s	571 req/s
⚪ orig: app-h--comments-ap-southeast-1	539 req/s	547 req/s
🔵 agg: app-h--comments-ap-southeast-1	536 req/s	544 req/s
🔵 agg: app-d--production-ap-southeast-1	509 req/s	513 req/s
⚪ orig: app-d--production-ap-southeast-1	464 req/s	480 req/s
🔵 agg: app-h--production-ap-southeast-1	452 req/s	463 req/s
⚪ orig: app-h--production-ap-southeast-1	447 req/s	454 req/s
🔵 agg: app-h--ai-ap-southeast-1	245 req/s	246 req/s

Every workload is drawn twice, aggregated over original. The pairs sit on top of each other. Compare **totals**, not pixel-for-pixel curves — the two lanes are sampled on different cadences, so they will never trace exactly.

We thought we were done

Until we sent the same data to a second backend… 😭



The 1-second resolution trap

OTel gateway · millisecond timestamps

gateway pod A
12:00:00.2 +30

gateway pod B
12:00:00.5 +30

gateway pod C
12:00:00.8 +20

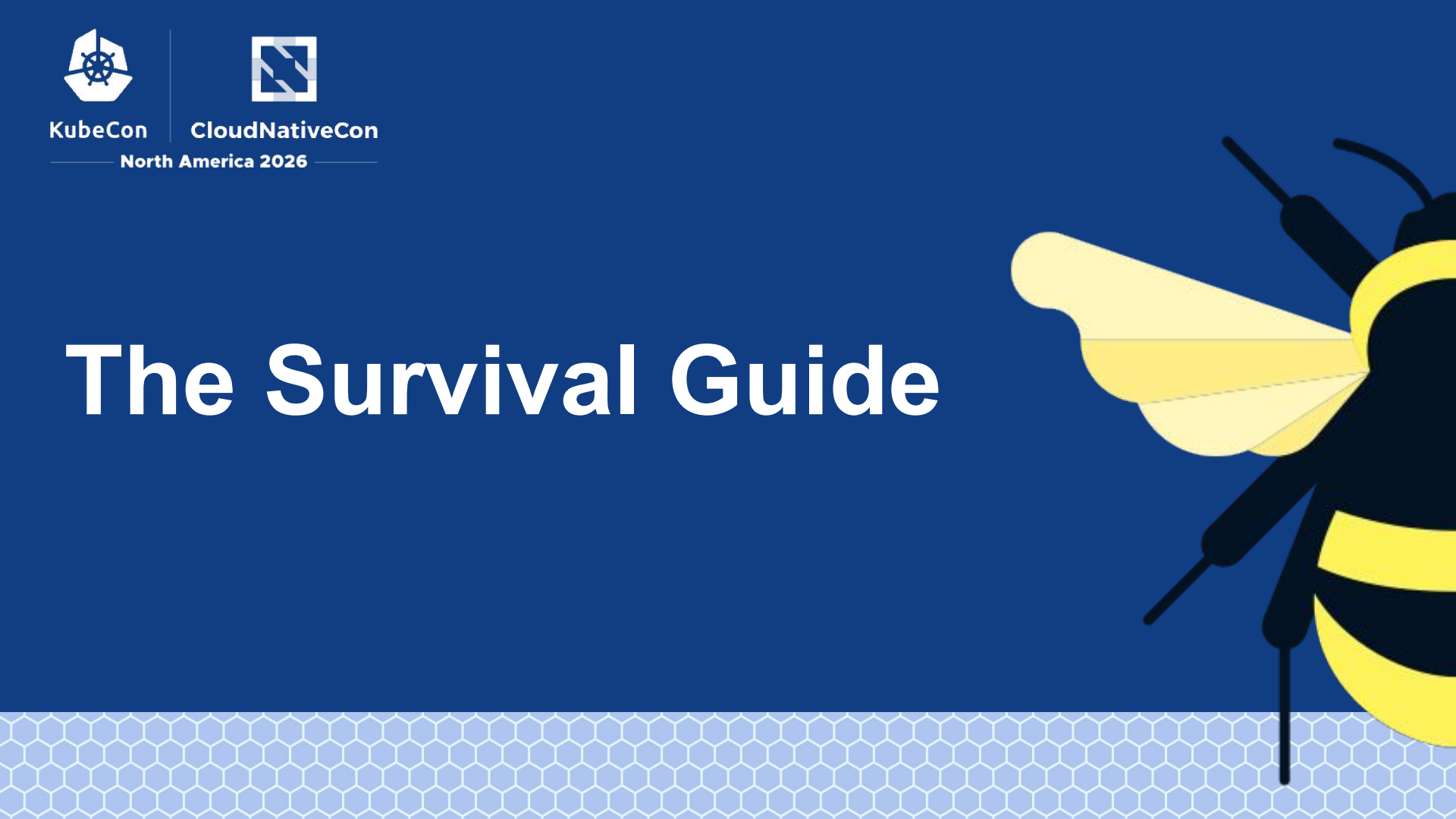


1-second resolution
backend slot 12:00:00
same second: last write wins



sent 80
stored 20

Impact: the backend read 2.79× low on our busiest cluster, 5.7× low for a 345-pod service.
Status: not closed yet. Testing one writer per series (gateway pod label), one point per interval (intervalprocessor), and a backend-side change.


KubeCon
CloudNativeCon
North America 2026
The Survival Guide

What to take home

1. Aggregate in-cluster, not in vendor UI
   - ✓ Keep configs in Git: reviewable, versioned, and portable.
2. Master the two layers
   - ✓ Resource attributes vs. datapoint attributes dictate how OTel merges.
3. Two-stage aggregation
   - ✓ DaemonSet collapses per node; Gateway collapses cluster-wide.
4. cumulativetodelta must run FIRST
   - ✓ Discarding pod identity also discards the cumulative baseline.
5. Beware silent inflation
   - ✓ rate() treats drops as resets, inflating numbers instead of firing alerts.
6. Migrate queries to sum_over_time
   - ✓ Swap out rate() and audit every downstream alert.
7. Check vendor timestamp resolution
   - ✓ Prevent collisions with one writer per series and one point per interval.

Two-Stage OTel Aggregation: Taming Istio Cardinality at 10k-Pod Scale

Give us feedback!

KubeCon CloudNativeCon North America 2026

linkedin.com/in/evelyn-chen-56b43853