KubeCon CloudNativeCon India 2026

#KubeCon #CloudNativeCon

Who Watches the Watchers?

From Closed Observability to Open Control at Scale

Aditi · Madhu · Sandeep

June 18, 2026

Aditi Gupta

Software Engineer II at JioHotstar (formerly Disney+ Hotstar)

Madhu

Software Engineer II at Adobe

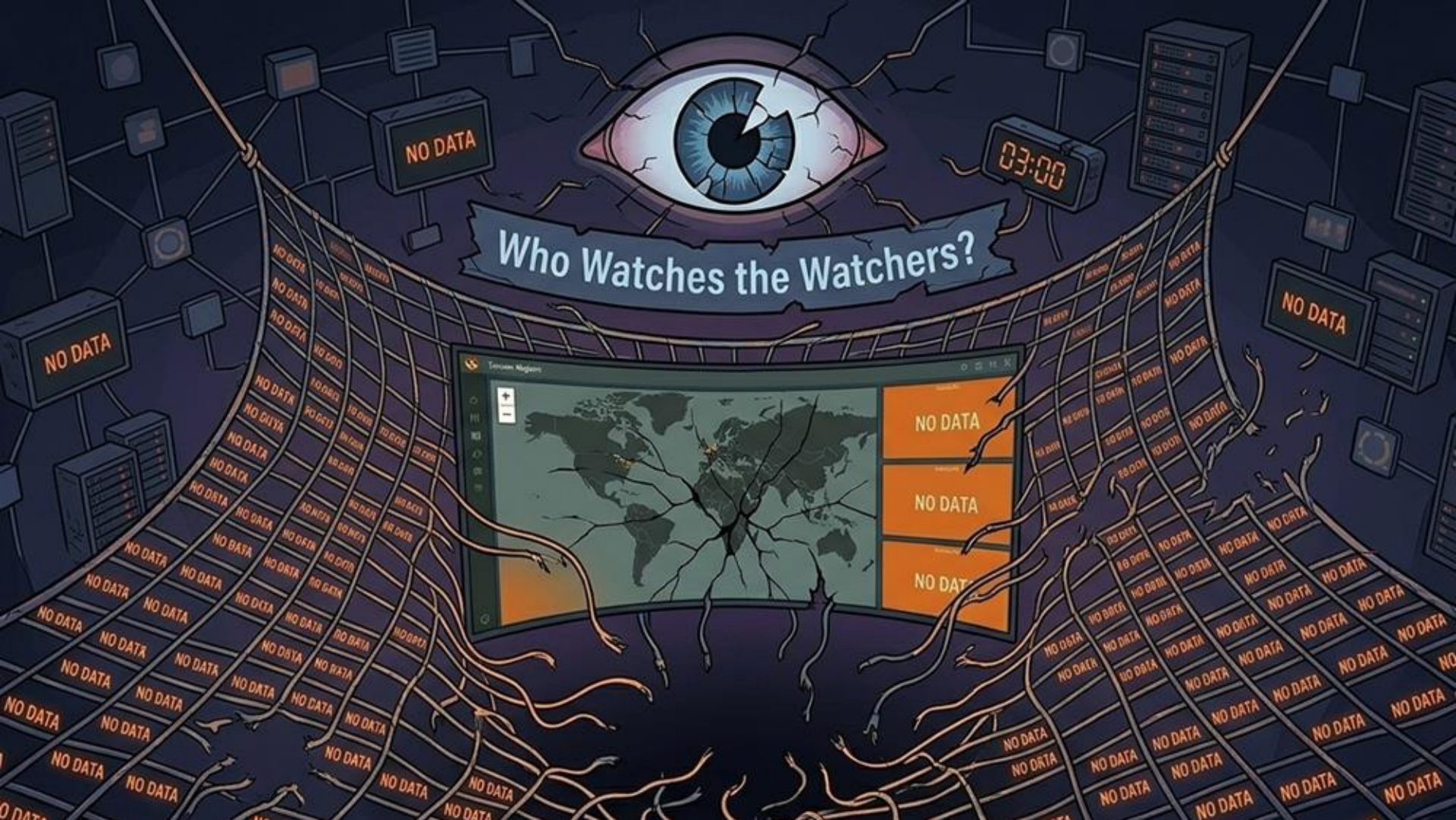
Sandeep Kanabar

Lead Software Engineer at Gen (formerly NortonLifeLock)

Co-chair, Merge Forward DHHWG

Who Watches the Watchers?


docker
aws
aws
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
03:00


Who Watches the Watchers?
NO DATA
03:00
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA

Dashboards go blank during an incident !!??

What We'll Cover - After your safety Net goes blank

01
The Problem
When your safety net goes blank

02
Why Dashboards Die
Cardinality bombs ·Block duration ·Retention Period

03 SOLUTION
Traffic Shaping
OTel Collector as a smart pipeline

04 SOLUTION
Defusing Cardinality
Prometheus recording rules that save memory

05 SOLUTION
Backpressure & Limits
Surviving 10x traffic spikes

06
Results & Mental Model
87.5% memory reduction — how we think now

Where This Talk Comes From

The Origin Story

A GDPR pipeline. A company acquisition.
A system I didn't build.

Where This Talk Comes From

The Origin Story

A GDPR pipeline. A company acquisition.
A system I didn't build.

10,000 msgs/day → 2,000,000 overnight.
20+ downstream systems. 100× load for 5 days.

The pipeline blew up.
And so did the observability.

The observability blew up too.

We found out

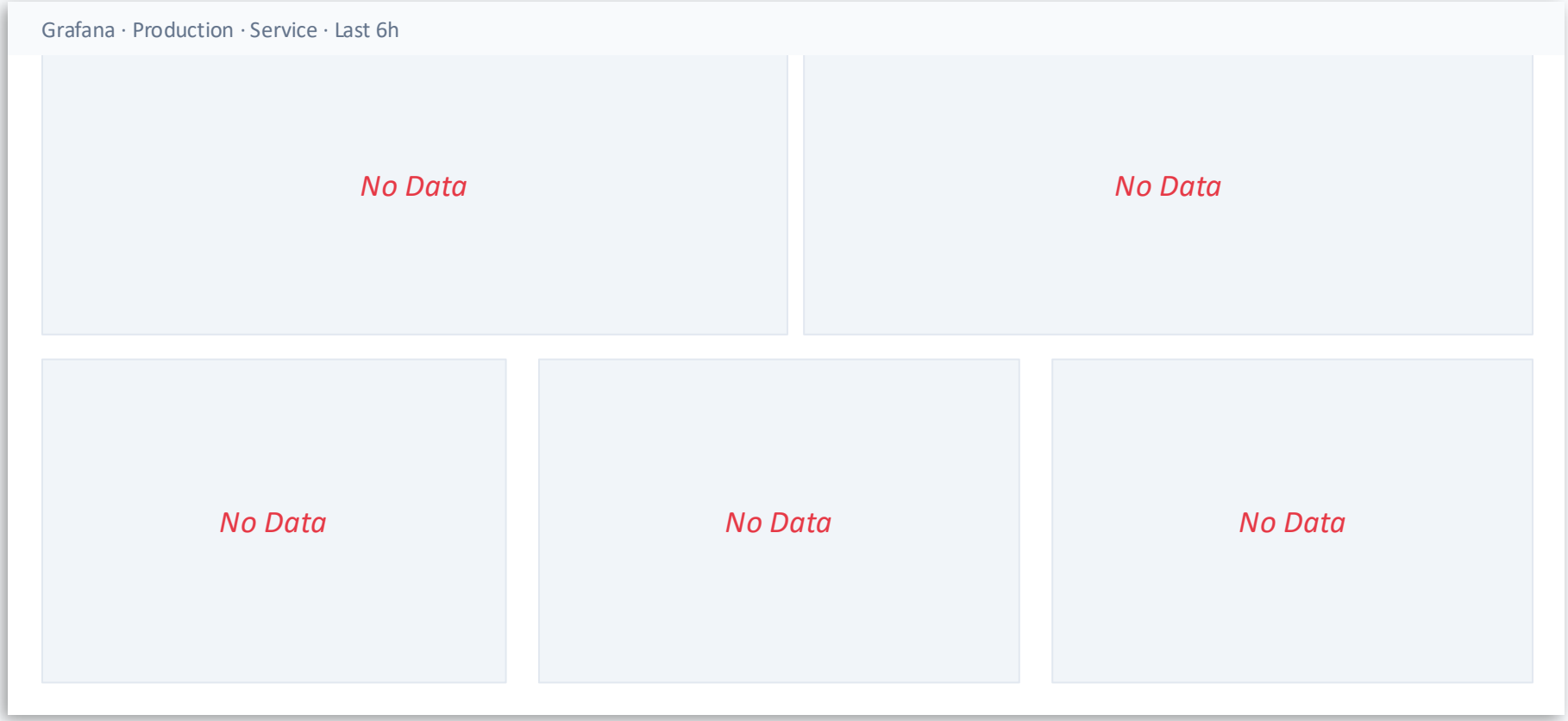
from a customer.

Not from an alert.
Not from a dashboard.

That's when I started asking: who watches the watchers?

3 AM.

Pager fires. You open the dashboard. MONITORING OFFLINE



The monitor is down.

Then Scale Hit.

10×

Traffic spikes

OOM

Prometheus crashes

Blind

During incidents

↑↑↑

Cardinality explosion

We didn't have too much data. We had too little control.

2M+	16 GB	124.8%	12 GB	500ms+
Active Series	Peak Memory	Over Limit	Index in RAM	Query Latency

WHY DASHBOARDS DIE

3 Failure Modes

That Took Down Prometheus

01

Cardinality Explosion

02

Block Duration

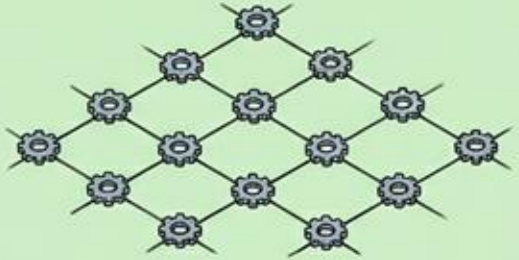
03

Memory Retention Policy

Failure Mode 1

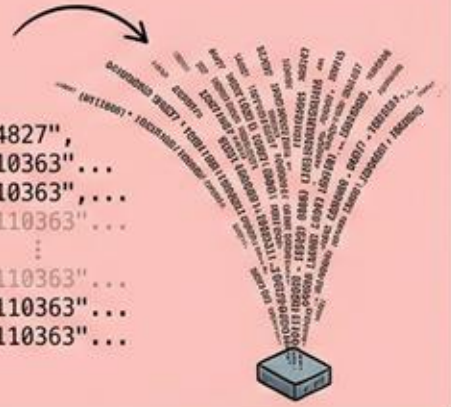
METRIC EXPLOSION: CARDINALITY ALERT

THE SAFE LABELS (LOW VARIANCE)



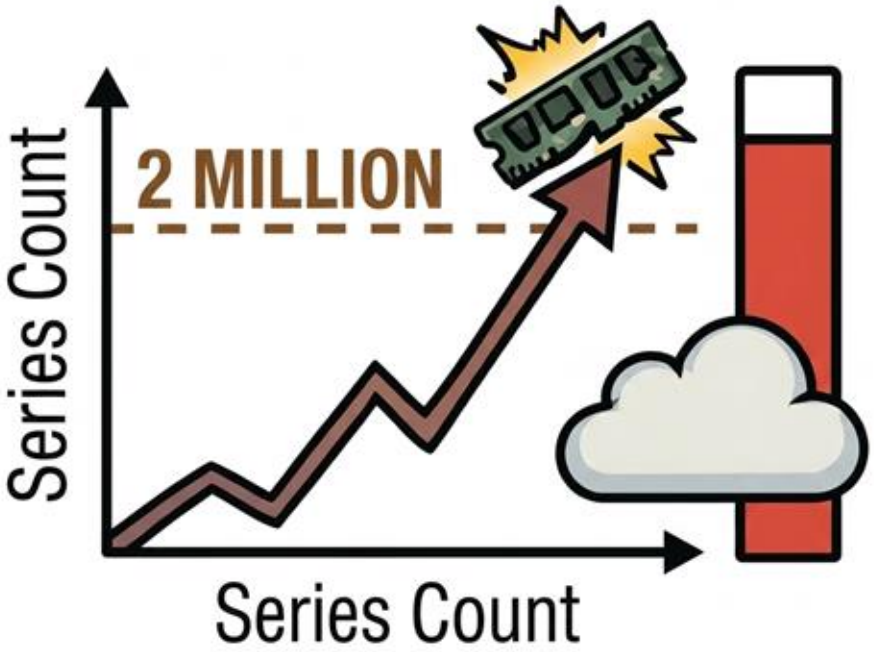
- **Service Name:**
  service="billing",
  service="api"
- **HTTP Method:**
  method="GET",
  method="POST"

THE HIGH-VARIANCE TRAP



- **USER ID:**
  user_id="94827",
  user_id="110363"...
  user_id="110363",...
  "110363"...
  ⋮
  "110363"...
  user_id=""110363"...
  user_id=""110363"...

TOTAL TIME-SERIES: (Overloading Prometheus)



PRODUCTION REALITY

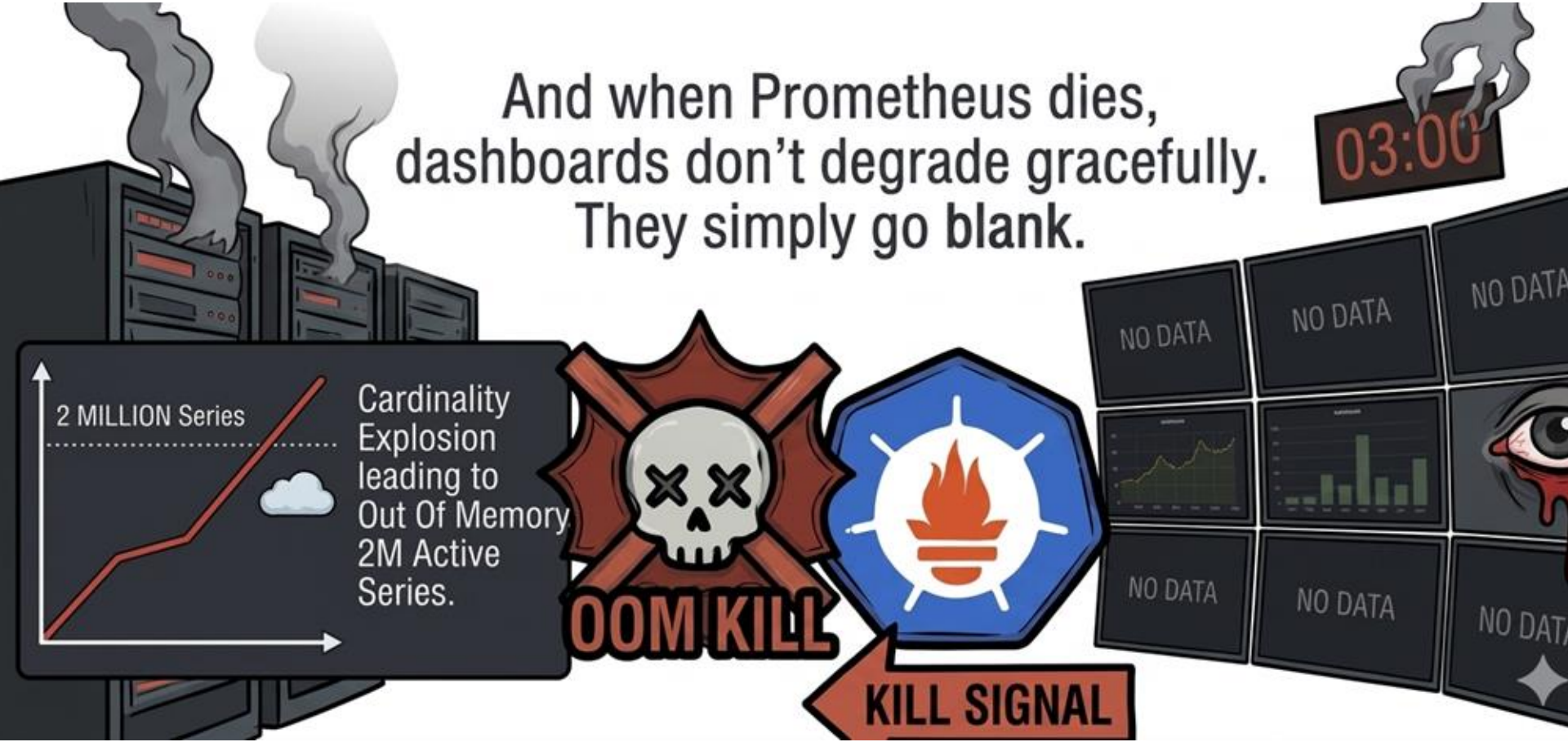
2,000,000+
Active series

50K+
Unique label pairs

12 GB
In-memory index

16 GB
Peak memory used

124.8%
Over memory limit → OOM kill


And when Prometheus dies,
dashboards don't degrade gracefully.
They simply go blank.
03:00
2 MILLION Series
Cardinality Explosion leading to Out Of Memory 2M Active Series.
OOM KILL
KILL SIGNAL
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA
NO DATA

Block Duration Mismatch

Three settings. One invisible problem.

01

Block Duration

02

Block Size

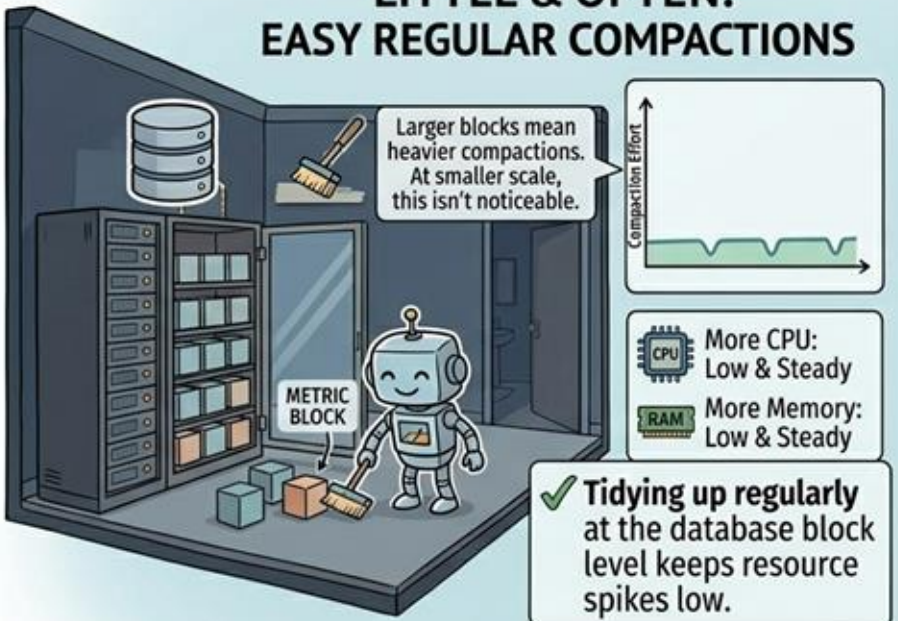
03

Compaction Freq.

COMPACTION EFFORT: A DATABASE ANALOGY

Think of metric block compactions like room tidying.

LITTLE & OFTEN: EASY REGULAR COMPACTIONS

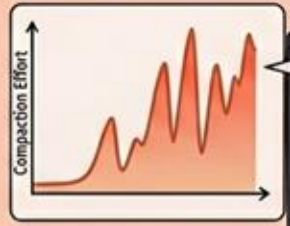


CPU More CPU: Low & Steady

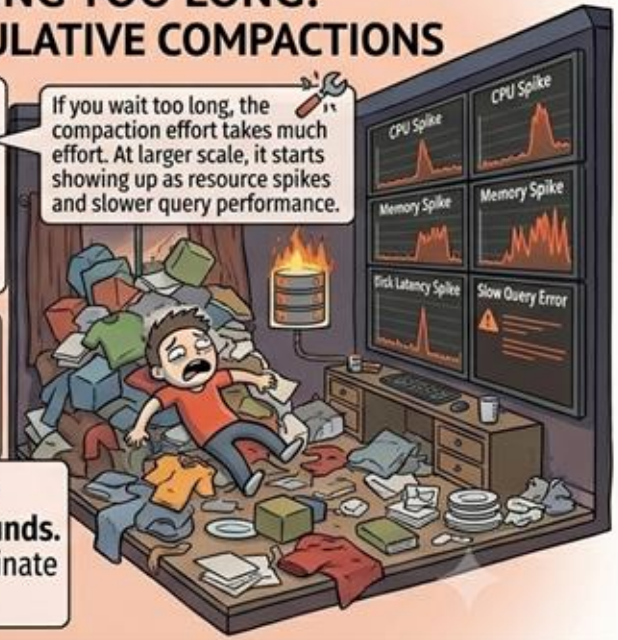
RAM More Memory: Low & Steady

✓ **Tidying up regularly** at the database block level keeps resource spikes low.

WAITING TOO LONG: HEAVY CUMULATIVE COMPACTIONS



If you wait too long, the compaction effort takes much effort. At larger scale, it starts showing up as resource spikes and slower query performance.



CPU More CPU: Massive Spikes

RAM More Memory: Massive Spikes

! **System resource pressure compounds.** Compactions dominate and queries slow.

SAME DATABASE. DIFFERENT AMOUNT OF WORK AT ONE TIME. Prometheus

Failure Mode 3 — Retention Nobody Questioned !!!

UNDERSTANDING THE TERMS

LOCAL RETENTION
How long Prometheus keeps data on local disk.

LOCAL DISK USAGE
Every extra day retained means more disk — and disk gets loaded into RAM.

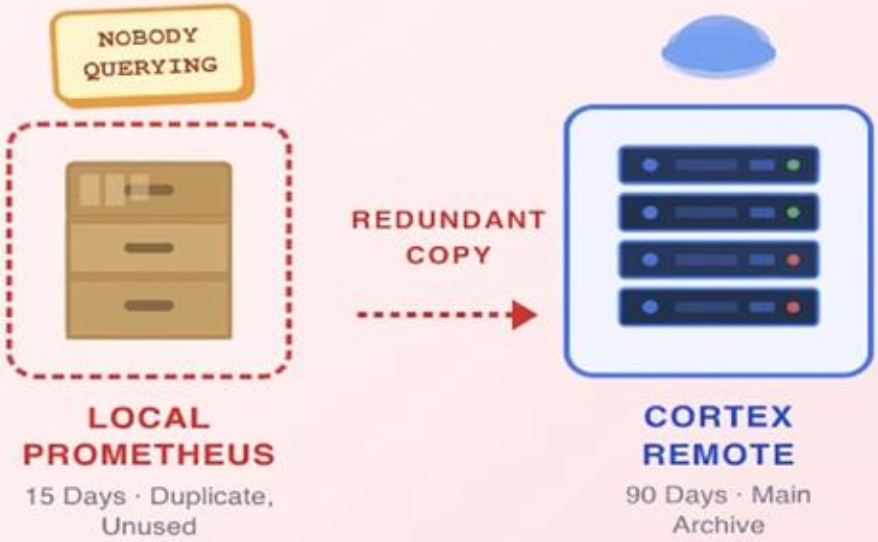
RETENTION STRATEGY
Your policy for when to flush local data to a remote store like Cortex.

THE PROBLEM:
WASTED RESOURCES

We were remote-writing everything to Cortex — 90-day history. But Prometheus was also holding 15 days locally. Same data, twice, in RAM.

16 GB of disk space. No query ever touched it.

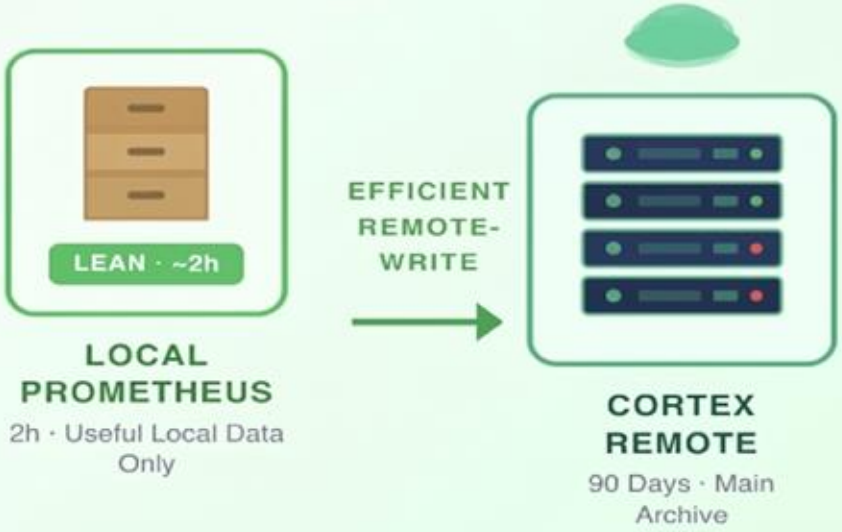
⚠ DUPLICATE DATA ARCHITECTURE



📈 High Disk & Memory Cost

Duplicate storage · Wasted RAM · No query benefit

✓ OPTIMIZED ARCHITECTURE



✅ Optimal Performance & Lower Cost

Tiered retention · Minimal local RAM · Full history in Cortex

Treat Telemetry as a First-Class Distributed System

The Vertical Scaling Trap — The Wrong Fix First

We gave it more RAM. It bought us time. It solved nothing.

If the underlying problem is cardinality or block duration, more RAM just delays the next OOM.

What we did (The Cycle)

- ↑ Pod memory limit
- Hit limit again
- ↑ More memory
- ***Repeat...***

What it cost

- More expensive pods
- Temporary relief only
- False confidence
- **Root cause ignored**

What we should have done →

Rebuilding with Control — Four Production Fixes

01

Active Traffic Shaping

OTel Collectors

- Batching
- Tail Sampling
- Memory Limiter

02

Cardinality Firewall

Prometheus

- Recording Rules
- Label Relabelling
- Retention Tiering

03

TSDB Surgery

Prometheus Config

- Block Duration
- WAL Compression
- Retention

04

Back Pressure & Limits

Resilience Layer

- Queue Limits
- Retry + Backoff
- Prometheus Remote-Write Tuning

Fix 01 Active Traffic Shaping · Optimizing collector loads

Batching

Groups spans/metrics into batches before forwarding

batch.yaml

```yaml
batch:
  timeout: 1s
  send_batch_size: 1024
```

Tail Sampling

Samples traces AFTER full trace assembled. 100% errors kept.

tail_sampling.yaml

```yaml
processors:
  tail_sampling:
    decision_wait: 10s
    policies:
      - name: errors-policy
        type: status_code
        status_code: ERROR
      - name: slow-traces
        type: latency
        threshold_ms: 500
```

Memory Limiter

Enforces a hard memory ceiling on the collector process

memory_limiter.yaml

```yaml
processors:
  memory_limiter:
    limit_mib: 512
    spike_limit_mib: 128
    check_interval: 5s
```

Fix 02 — Defusing Cardinality Bombs · Prometheus

Recording Rules

Pre-aggregate high-cardinality metrics into stable summaries.

Before: query scans 2M series → timeout
After: reads 1 pre-aggregated series → <50ms

```
groups:
- name: http_aggregates
  rules:
  - record: job:http_requests:rate5m
    expr: sum by(job, status)(rate(http_requests_total[5m]))
```

Label relabelling — drop toxic labels at scrape time:

```
metric_relabel_configs:
- source_labels: [__name__]
  action: labeldrop
  regex: uri|client_id|container_id
```

Series Limits

Hard cap per scrape target.
sample_limit: 10000 per job.
Exceeded → Prometheus drops & alerts. Not silently.

Tiered Storage Strategy

Raw high-cardinality → 2h local TTL
Aggregated recording rules → long TTL in Cortex
Best of both worlds.

Fix 03 — TSDB Surgery · Prometheus Config

Block Duration 2 hours → 15 minutes

Compaction spikes: 20 GB → 2.5 GB · 87.5% reduction

```
minBlockDuration: 15m  # was 2h
maxBlockDuration: 15m
```

Retention 15 days / 16 GB → 2h + Cortex

16 GB of unneeded local history removed

```
retention: 2h          # was 15 days
retentionSize: 1GB
```

WAL Compression Off / unset → walCompression: true

40–50% WAL size reduction. One line. Check yours.

```
walCompression: true
```

These three settings interact. Combined with cardinality fixes: total memory 16 GB → 2 GB (87.5%).

Fix 04 — Back Pressure & Limits · Resilience Layer

App / Service → OTel Collector (Gatekeeper) → Prometheus / Loki (Backend) → Grafana (Safe)

Queue Limits

Set explicit queue_size on OTel exporters.

```
sending_queue:
  num_consumers: 10
  queue_size: 5000
```

Retry + Exponential Backoff

Prevent thundering herd. Spread retries after backend recovery.

```
retry_on_failure:
  initial_interval: 5s
  max_interval: 30s
  max_elapsed_time: 300s
```

Prometheus Remote-Write Tuning

Prevents write-path pile-up during spikes.

Tune to match your Cortex ingestion throughput.

```
remote_write:
       queue_config:
              capacity: 10000
              max_shards: 200
              batch_send_deadline: 5s
```

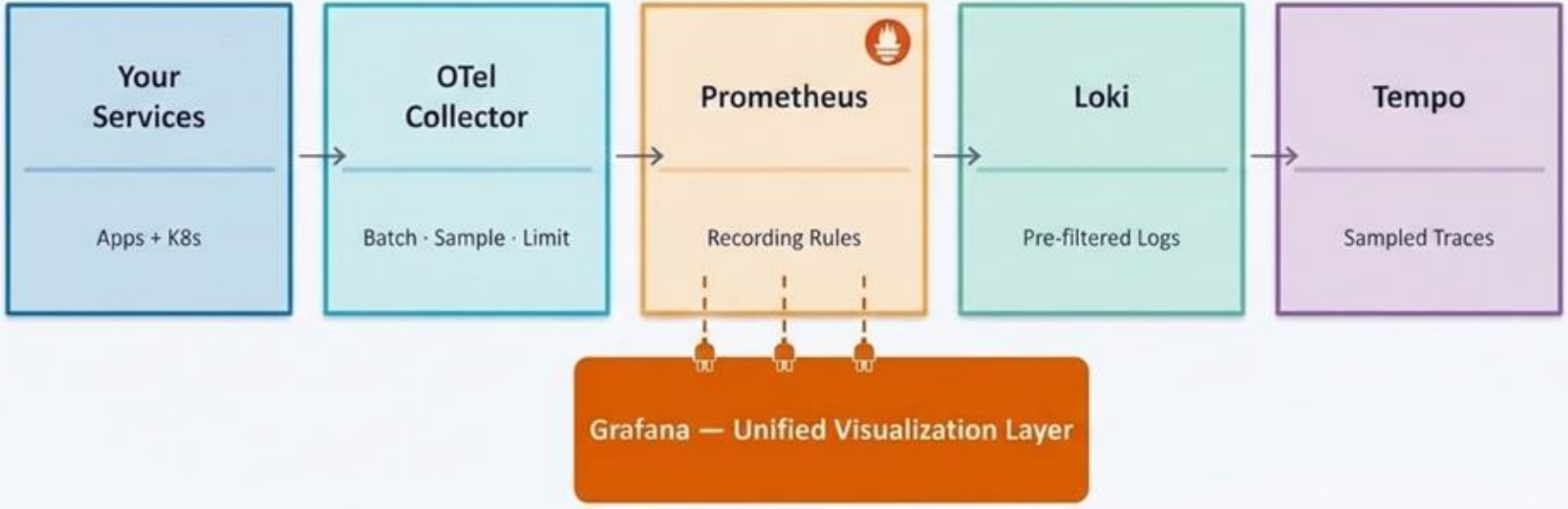
The Numbers Don't Lie

Metric	Before	After
Prometheus OOM crashes	Recurring	Zero
Memory usage	16 GB (124% limit)	2 GB (50% limit)
Active series	2,000,000+	848,320
In-memory index	12 GB	500 MB
Compaction spike	20 GB every 2h	2.5 GB every 15m
Query latency	500ms+	< 50ms
Dashboard load (peak)	Timeout / N/A	< 2s
10× spike handling	Brownout	Controlled degradation
Incident MTTR	Blind → hours	Visible → minutes

60 GB total RAM freed across 4 production + 4 staging regions — same pattern, same fixes.

The Fix:

Treat Telemetry as a First-Class Distributed System



Active Traffic Shaping

Batching + Tail Sampling

Cardinality Firewall

Recording Rules + Label Budgets

Backpressure & Limits

Queue Limits + Graceful Degradation

What Changed

The Mental Model Shift

BEFORE	NOW
Collect everything, figure it out later	**Pipeline-first thinking: design telemetry flow before choosing tools**
Labels are free — add as many as you want	**Cardinality budget: treat label cardinality like memory — spend it deliberately**
If the collector goes down, restart it	**Degrade gracefully: losing 10% of traces beats losing 100%**
Vertical scale when things break	**Design for failure: backpressure, limits, and circuit breakers at every layer**

Three Things to Take Home

1

Your pipeline IS a distributed system

Apply limits, backpressure, and graceful degradation to your telemetry stack — not just your applications.

2

Control > Collection

Stop collecting everything. Start controlling what you collect, what you keep, and what you precompute. Signal quality beats data volume every time.

3

Build for failure, not for sunshine

Your monitoring must survive a 10× traffic spike. If it doesn't, it's not a safety net — it's a liability waiting to activate.

KubeCon | CloudNativeCon
India 2026

Observability is a safety net.

But a safety net that fails under stress is just a very expensive false alarm.

We rebuilt ours to survive the storm.

Now you know how to rebuild yours.

Aditi · Madhu · Sandeep

KubeCon India · June 18, 2026 · #KubeCon #CloudNativeCon

KubeCon CloudNativeCon India 2026

“Who watches the watchers?”

Quis custodiet ipsos custodes?

— Juvenal, *Satires*

Aditi · Madhu · Sandeep

KubeCon India · June 18, 2026 · #KubeCon #CloudNativeCon