KUBECON + CLOUDNATIVECON NORTH AMERICA 2026 · SALT LAKE CITY

Beyond Dashboards

A Semantic Layer for AI-Driven Observability in Kubernetes



Pronnoy Goswami

Senior Software Engineer, NVIDIA

Thursday, November 12, 2026 · Grand Ballroom BDF

About me



Pronnoy Goswami

- Senior Software Engineer at NVIDIA
- Distributed systems, AI/ML infrastructure, and observability
- Previously built AI-assisted observability for a petabyte-scale, multi-region Kubernetes platform
- Writes the newsletter Distributed Bytes

FIND ME

LinkedIn **in/pronnoygoswami**

X **@pronnoygoswami**

GitHub **goswamipronnoy**

Views are my own and do not represent any current or former employer. No products are being pitched.

The 2 AM page

WHAT ONE MULTI-REGION PLATFORM PRODUCES

Tens of TB of logs every day

Millions of metric data points

Millions of distributed traces

Thousands of correlation IDs per minute

“We don’t have a data shortage problem. We have a data meaning problem.”

WHAT ON-CALL ACTUALLY DOES

- Correlates by hand across tools
- Depends on tribal knowledge
- Fights alert fatigue

Why raw OTel pipelines fail with LLMs

× Volume beats the context window

Minutes of logs from one busy namespace can exceed a model's context. Truncation silently drops the evidence.

× No schema, so the model guesses

It invents metric names, labels, and PromQL that look plausible and return nothing, or the wrong thing.

× Unbounded queries

A free-form query over a high-cardinality label can time out, or load your TSDB in the middle of the incident.

× No provenance

The answer cannot point to the query and data behind it, so on-call cannot verify it at 2 AM.

The model is rarely the problem. The interface between the model and the telemetry is.

The idea

Give the model a typed contract, not a firehose.

Model Context Protocol (MCP) is an open protocol that lets AI applications discover and call tools exposed by external systems, through schema-described interfaces.

- **OpenTelemetry** standardizes how telemetry is produced.
- **MCP** can standardize how AI consumes it.

Raw telemetry access vs a semantic layer

	Raw telemetry access	Semantic MCP layer
Who writes the query	The model writes PromQL	The model picks a signal from a catalog
Inputs	Any label, any time range	Enums and bounded windows in the schema
Output	Thousands of raw data points	Current vs baseline, in a few fields
Errors	Empty result; the model guesses again	“Unknown service. Call list_services first.”
Trust	No trail	Every answer carries the query behind it

A four-layer architecture

L1

Instrument & enrich

OTel SDKs and semantic conventions. The same context keys on every signal.

OpenTelemetry SDKs

L2

Collect & store

Collector adds Kubernetes metadata and keeps metric labels bounded.

OTel Collector · Prometheus / Mimir · Fluentd

L3

Semantic query layer

MCP server with typed, bounded tools that return summaries and provenance.

Model Context Protocol

L4

Agents

Investigate, correlate, and propose a root cause with evidence. Humans approve actions.

Any MCP-capable agent

Meaning is added at write time

Meaning is enforced at read time

Everything here is open source or an open standard, and every layer can be swapped.

L1 Enrich at the source

```yaml
processors:
  k8sattributes:
    extract:
      metadata:
        - k8s.namespace.name
        - k8s.deployment.name
        - k8s.pod.name
  resource:
    attributes:
      - key: deployment.environment.name
        value: prod
        action: upsert
  batch: {}

service:
  pipelines:
    metrics:
      receivers: [otlp]
      processors: [k8sattributes, resource, batch]
      exporters: [otlphttp]
```

OpenTelemetry Collector configuration (excerpt)

Semantic conventions are the vocabulary

service.name, k8s.namespace.name, deployment.environment.name. The agent will query by these names.

Propagate W3C trace context

Logs and spans share trace_id, so one ID joins signals across services.

Same keys on every signal

Metrics, logs, and traces all carry the same core context.

L2 Keep metric labels bounded

CONTEXT	AS A METRIC LABEL?	WHERE IT BELONGS
`service.name (job)`	Yes	Metrics, logs, and traces
`k8s.namespace.name`	Yes	Metrics, logs, and traces
`http.response.status_code`	Yes	Metrics and spans
`k8s.pod.name`	Careful	Churns on every rollout. Aggregate to deployment for agents.
`request_id, user_id, order_id`	Never	Spans and logs. Reach them from metrics via exemplars.

High-cardinality context lives on spans and logs. Metrics point to it through exemplars (trace_id).

L3 Design tools, not endpoints

Typed

Enums and bounded ranges live in the schema. Invalid calls fail fast.

Discoverable

list_services teaches the agent your vocabulary instead of letting it guess.

Summarized

Return current vs baseline and percent change, not 10,000 raw points.

Cited

Every response carries the exact query that produced it.

TOOL SURFACE

In the demo

```
list_services()
get_signal(service, signal)
```

Natural next tools

```
find_exemplar_traces()
get_k8s_events()
recent_deploys()
```

L3 Code: a typed tool over Prometheus / Mimir

```python
mcp = MCPServer("k8s-metrics-semantic")

# OTel semconv name after Prometheus OTLP translation
HTTP = "http_server_request_duration_seconds"
CATALOG = {
    "request_rate": lambda s: f'sum(rate({HTTP}_count{{job="{s}"}}[5m]))',
    # "error_ratio", "p99_latency_seconds": same pattern
}
Signal = Literal["request_rate", "error_ratio", "p99_latency_seconds"]

@mcp.tool()
async def get_signal(
    service: str,
    signal: Signal,
    window_minutes: Annotated[int, Field(ge=5, le=360)] = 30,
) -> SignalSummary:
    """Summarize one golden signal for a service: the last `window_minutes`
    compared with the window just before it."""
    if service not in await _known_services():
        raise ToolError(f"Unknown service '{service}'. "
                        "Call list_services first.")
    promql = CATALOG[signal](service)
```

1

A catalog, not PromQL

The model picks a signal name. Humans own the queries.

2

Schema does the guarding

Literal becomes an enum; Field(ge, le) becomes min and max.

3

Errors the agent can act on

ToolError text goes back to the agent, so it can self-correct.

Python MCP SDK 2.x (MCPServer, formerly FastMCP).

L3 Code: return meaning, not data points

```python
step = max(15, (2 * window_minutes * 60) // 120)  # cap at ~120 points
r = await client.get(
    "/api/v1/query_range",
    params={"query": promql, "start": start, "end": end, "step": step},
)
r.raise_for_status()
result = r.json()["data"]["result"]
values = result[0]["values"] if result else []

mid = end - window_minutes * 60
points = [(t, float(v)) for t, v in values]
points = [(t, v) for t, v in points if not math.isnan(v)]  # drop NaN
base = [v for t, v in points if t < mid]
curr = [v for t, v in points if t >= mid]

c, b = mean(curr), mean(base)
change = (c - b) / b * 100 if c is not None and b else None
return SignalSummary(
    service=service, signal=signal, current=c, baseline=b,
    change_pct=change, points=len(values), promql=promql,
)
```

```
{
  "service": "checkout",
  "signal": "error_ratio",
  "current": 0.1045,
  "baseline": 0.0041,
  "change_pct": 2443,
  "points": 121,
  "promql": "sum(rate(http_server_
    request_duration_seconds_
    count{job=\"checkout\",
    ...}[5m])) / ..."
}
```

What the agent reads. From the demo run; values rounded and promql shortened for the slide.

Mimir: same code. Point PROM_URL at /prometheus and set the X-Scope-OrgID header.

Demo: an investigation on a live stack



Recorded demo

WHAT IS RUNNING

- **3 sample services**
  OpenTelemetry SDK
- **OTel Collector**
  OTLP in, metrics and traces out
- **Prometheus 3.5**
  native OTLP ingestion
- **MCP server**
  the code you just saw
- **Scripted MCP client**
  agent-style triage, no LLM

Fault: payments +1 s

What the agent saw

```
$ ./demo.sh investigate
1. Discovered tools: list_services, get_signal
2. Services reporting metrics: checkout, inventory, payments
3. Golden signals, last 15 min vs previous 15 min:
   !! checkout   error_ratio           +2443%
   !! checkout   p99_latency_seconds    +287%
      checkout   request_rate             0%
      inventory  error_ratio            -18%
      inventory  p99_latency_seconds     +2%
      inventory  request_rate             0%
      payments   error_ratio            -24%
   !! payments   p99_latency_seconds    +194%
      payments   request_rate           -10%
4. Anomalies, largest change first (with evidence):
   - checkout.error_ratio: 0.00411 -> 0.105 (+2443%)
   - checkout.p99_latency_seconds: 0.249 -> 0.964 (+287%)
   - payments.p99_latency_seconds: 0.149 -> 0.439 (+194%)
```

Output from the demo run. Synthetic services and traffic. promql lines omitted.

Tools are discovered at runtime

No prompt engineering about metric names.

Ranking finds the symptom

checkout errors move the most, so they rank first.

The cause is upstream

payments is slow, not failing. A trace shows it; that is L4's job.

L4 Agents that investigate like an SRE

1
Alert fires
from Alertmanager

2
Scope the blast radius
`list_services, namespace`

3
Compare signals to baseline
`get_signal`

4
Correlate changes
deploys, k8s events

5
Follow exemplars to traces
symptom → cause

6
Propose a root cause, with cited queries
a human reviews it

GUARDRAILS

Read-only tools by default
The agent investigates. It does not remediate.

Budgets per investigation
A cap on tool calls and wall-clock time.

Humans approve every action
The RCA is a proposal with evidence, not a decision.

Production lessons: cardinality and latency

Cardinality

Never offer free-form group-by

Expose an enum of allowed dimensions. Agents will happily group by pod.

Back the catalog with recording rules

Hot agent queries become cheap, fast, and predictable.

Aggregate away churn

Pod names change on every rollout. Agents query deployments.

Latency

Agents call tools in loops

A 3 s query × 20 tool calls is a full minute of waiting. Set hard timeouts.

Cache what changes slowly

Service lists and metric metadata can be cached for minutes.

Summarize inside the server

Fewer tokens in means faster, cheaper reasoning and fewer hallucinations.

What changed in production

8 min

Average incident resolution, down from 45 min

82%

MTTR reduction

65%

Reduction in alert fatigue

3.2×

Faster detection

WHAT MADE THE DIFFERENCE

- Correlation moved from incident time to write time, so on-call stopped joining signals by hand
- One place to ask questions instead of switching between tools

Measured in one multi-region production environment, in a previous role. Results depend on how far you get with L1.

When does agentic observability reduce MTTR?

It helps when

- Signals share consistent context (L1 is done)
- Incidents span several services and signals
- Failure patterns repeat
- On-call time goes into correlation, not the fix

Not yet, when

- service.name and trace context are inconsistent
- The alert is single-service with a known runbook
- There is no baseline yet (brand-new services)
- You cannot grant read access safely

Measure time to the first correct hypothesis, not how satisfied people are with the chat.

Start on Monday

1

Enrich one critical service

Semantic conventions plus k8sattributes

2

Wrap your TSDB in 2–3 typed tools

Catalog, bounds, provenance

3

Shadow real incidents

Compare the agent's RCA with the human one

4

Add budgets and caching

Recording rules, timeouts, cached metadata

5

Expand service by service

Let incidents choose the next service

Anti-pattern: designing the perfect context schema up front.

Let real incidents show you which fields matter. Measure what on-call uses, add what is missing, remove what is not used.

Three takeaways

1. **Raw telemetry is the wrong interface for an LLM.**

2. **Add meaning at write time with OpenTelemetry. Enforce it at read time with typed MCP tools.**

3. **Bound everything: labels, windows, outputs, and budgets.**

“The observability problem is fundamentally a data problem. Solve it at the data layer.”

Thank you

Questions?

LinkedIn **in/pronnoygoswami**

X **@pronnoygoswami**

GitHub **goswamipronnoy**

RATE THIS SESSION



Beyond Dashboards: A Semantic Layer for AI-Driven Observability in Kubernetes

Scan to leave feedback