From Signal to Insight

Building Queryable Observability with the Model Context Protocol

Pronnoy Goswami
Software Engineer @ Workday

About Me

- Software Engineer at Workday. Previously at Microsoft, PayPal, and McKinsey & Company.

- Experienced with Distributed Systems, AI, Cloud Infrastructure, AI/ML Infra, and Observability.

- Newsletter Author: Distributed Bytes with Pronnoy

linkedin.com/in/pronnoygoswami/

x.com/pronnoygoswami

github.com/goswamipronnoy

Why Should You Care?

The 2 AM Wake-up Call: An On-Call Nightmare

The Data Avalanche

- Tens of TB of logs daily
- Millions of metric data points
- Millions of distributed traces
- Thousands of correlation IDs/min

The Engineer's Reality

- Manual correlation across systems
- Tribal knowledge dependency
- Alert fatigue
- Delayed root cause analysis

"We don't have a data shortage problem—we have a data meaning problem."

Why Modern Observability Fails?

50% of organizations report siloed telemetry data. Only 33% achieve unified views across metrics, logs, and traces. – *New Relic's 2023 Observability Forecast Report*

Metrics

What is happening?

Logs

Why it happened?

Traces

Where it happened?

The Core Problem

Without a consistent thread of context, debugging and manual correlation is hard!

The Paradigm Shift

Traditional Approach

- Correlation at analysis time (during incidents)
- Slower MTTx
- Higher alert fatigue
- Reduced developer productivity.

AI-First Approach with MCP

- Correlation at telemetry creation time
- Faster MTTx
- Lower alert fatigue
- Increased developer productivity.

Model Context Protocol: A Data Pipeline for AI

Model Context Protocol (MCP) as an open standard that allows developers to create a secure two-way connection between data sources and AI tools. – [Anthropic]

Contextual ETL for AI

Standardizes context extraction from multiple data sources

Structured Query Interface

AI readable, transparent access to enriched data layers

Semantic Data Enrichment

Embeds meaningful context directly into telemetry signal

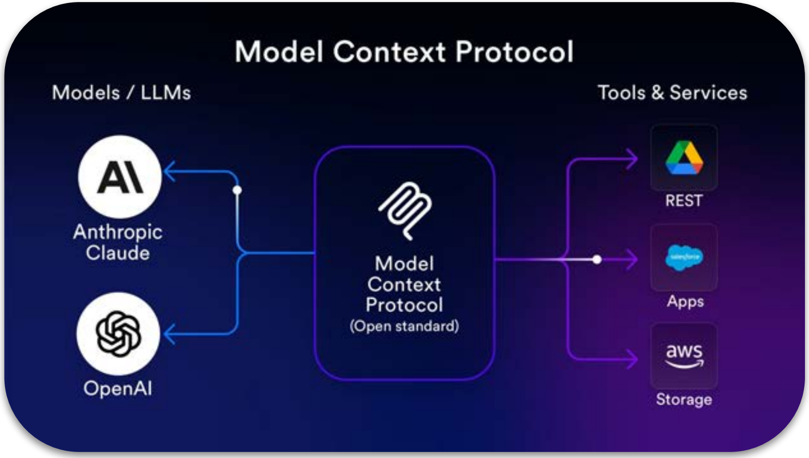
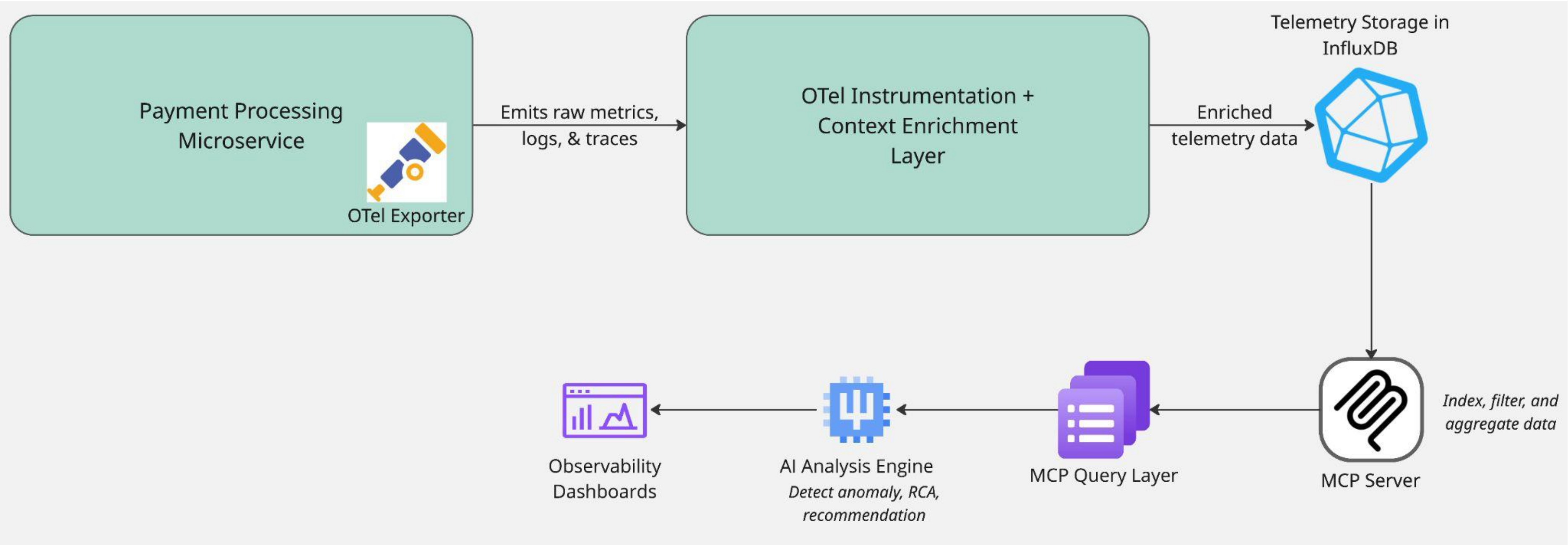


Image Credits: [Lightflows]

Proposed Architecture

A New Three-Layer Architecture



Shifts observability from reactive problem-solving to proactive insights.

Implementation Deep-Dive

Layer 1 : Context Enrichment

```python
def process_checkout(user_id, cart_items, payment_method):
    """Simulate a checkout process with context-enriched telemetry."""

    # Generate correlation id
    order_id = f"order-{uuid.uuid4().hex[:8]}"
    request_id = f"req-{uuid.uuid4().hex[:8]}"

    # Initialize context dictionary that will be applied
    context = {
        "user_id": user_id,
        "order_id": order_id,
        "request_id": request_id,
        "cart_item_count": len(cart_items),
        "payment_method": payment_method,
        "service_name": "checkout",
        "service_version": "v1.0.0"
    }

    # Start OTel trace with the same context
    with tracer.start_as_current_span(
        "process_checkout",
        attributes={k: str(v) for k, v in context.items()}
    ) as checkout_span:

        # Logging using same context
        logger.info(f"Starting checkout process", extra={"context": json.dumps(context)})

        # Context Propagation
        with tracer.start_as_current_span("process_payment"):
            # Process payment logic...
            logger.info("Payment processed", extra={"context": json.dumps(context)})
```

- Generate correlation IDs for context propagation.
- Initialize the context dictionary to enrich spans.

Key Idea: Every telemetry signal contains the same core contextual data.

The Impact of Context Enrichment

A Sample Debugging Scenario

Without Context ❌

```
2025-01-15 14:23:41 ERROR
Payment failed

2025-01-15 14:23:42 WARN
Timeout on service call

2025-01-15 14:23:43 ERROR
Transaction rolled back
```

Engineer's burden

- Which user?
- Which order?
- What service failed?
- How to correlate these logs?

With Trace Context ✅

```
2025-01-15 14:23:41 ERROR
context: {
  "request_id": "req-a1b2c3d4",
  "order_id": "order-xyz789",
  "user_id": "usr_12345",
  "service": "payment"
}
Payment processing timeout

// All related logs share
// same request_id
```

Immediate clarity

- Instant correlation via request_id
- Service identification
- User impact assessment
- Automated trace assembly

Layer 2: MCP Server – Structured Query Interface

```python
@app.post("/mcp/logs", response_model=List[Log])
def query_logs(query: LogQuery):
    """Query logs with specific filters"""
    results = LOG_DB.copy()

    # Apply contextual filters
    if query.request_id:
        results = [log for log in results if log["context"].get("request_id") == query.request_id]

    if query.user_id:
        results = [log for log in results if log["context"].get("user_id") == query.user_id]

    # Apply time-based filters
    if query.time_range:
        start_time = datetime.fromisoformat(query.time_range["start"])
        end_time = datetime.fromisoformat(query.time_range["end"])
        results = [log for log in results
                   if start_time <= datetime.fromisoformat(log["timestamp"]) <= end_time]

    # Sort by timestamp
    results = sorted(results, key=lambda x: x["timestamp"], reverse=True)

    return results[:query.limit] if query.limit else results
```

1

2

3

Indexing

Efficient lookups via context fields

Filtering

Precise data segregation

Aggregation

Statistical Computation

Layer 3: AI-Driven Analysis Engine

```python
def analyze_incident(self, request_id=None, user_id=None, timeframe_minutes=30):
    end_time = datetime.now()
    start_time = end_time - timedelta(minutes=timeframe_minutes)
    time_range = {"start": start_time.isoformat(), "end": end_time.isoformat()}

    # Fetch relevant telemetry based on context
    logs = self.fetch_logs(request_id=request_id, user_id=user_id, time_range=time_range)

    services = set(log.get("service", "unknown") for log in logs)

    metrics_by_service = {}
    for service in services:
        for metric_name in ["latency", "error_rate", "throughput"]:
            metric_data = self.fetch_metrics(service, metric_name, time_range)

            # Calculate statistical properties
            values = [point["value"] for point in metric_data["data_points"]]
            metrics_by_service[f"{service}.{metric_name}"] = {
                "mean": statistics.mean(values) if values else 0,
                "median": statistics.median(values) if values else 0,
                "stdev": statistics.stdev(values) if len(values) > 1 else 0,
                "min": min(values) if values else 0,
                "max": max(values) if values else 0
            }

    anomalies = []
    for metric_name, stats in metrics_by_service.items():
        if stats["stdev"] > 0:  # Avoid division by zero
            z_score = (stats["max"] - stats["mean"]) / stats["stdev"]
            if z_score > 2:  # More than 2 standard deviations
                anomalies.append({
                    "metric": metric_name,
                    "z_score": z_score,
                    "severity": "high" if z_score > 3 else "medium"
                })
```

- Define analysis time window
- Extract relevant telemetry based on the current context.
- Calculate statistical properties to capture anomalies and AI recommendations.

Context enables automatic cross-signal analysis without manual intervention.

What is the Impact?

Real-World Impact: From Hours to Minutes

Before AI-Enhanced Telemetry

45 min

Average Incident Resolution

- Manual log correlation
- Multiple tool context switches
- Tribal knowledge required
- High cognitive load

After the AI-Enhanced Telemetry

8 min

Average Incident Resolution

- Automatic correlation
- Single-pane-of-glass (unified) analysis
- Context-driven insights
- AI-suggested and enhanced RCAs

↓82%
MTTR Reduction

↓65%
Alert Fatigue

↑3.2x
Faster Detection

Is The Complexity Worth It?

"This seems like a lot of overhead. Can't we just use better dashboards?"

The Hidden Costs You're Already Playing

- **Engineer Time:** 15-20 hrs/week on-call load
- **Alert Fatigue:** ~60-70% of alerts are unactionable
- **Context Switching:** 23 minutes to refocus after an interruption
- **Tribal Knowledge:** Critical dependencies on certain individuals

The Investment Payoff

- **One-time setup:** 2-3 weeks investment
- **Incremental adoption:** Start with critical services
- **Permanent gains**: Replicate to every service
- **ROI timeline:** Break-even in 6-8 weeks

Is The Complexity Worth It?

Key Insight

The complexity you add at generation time eliminates exponentially more complexity at analysis time.

What Are the Key Takeaways?

Insight #1: Start with Context Enrichment

Begin Small, Think Big

Don't implement the full 3-layered architecture at one. Start by enriching one critical service.

Minimum Viable Context

Identify crucial "tags" that solve majority of the correlation use cases.

```
{
  "request_id": "...",
  "user_id": "...",
  "service": "...",
  "environment": "..."
}
```

1. Add context dictionary to your logging setup
2. Propagate context through service calls
3. Include context in OTel spans
4. Validate context appears in all signals
5. Expand to other services

Insight #1: Start with Context Enrichment

Week 1 Goal

Pick one high-traffic service and add basic context enrichment.

Insight #2: Build Query-able Interfaces

Make Data AI-Ready

Even without full MCP implementation, create structured APIs over your telemetry.

Simple REST API Pattern

```
GET /api/logs?request_id=xxx
GET /api/metrics?service=payment&time_range=...
GET /api/traces?request_id=xxx

# Returns structured, filterable data
# AI can query programmatically
```

Benefits

- Programmatic access
- Consistent Interfaces
- AI-friendly formats

Insight #2: Build Query-able Interfaces

Quick Win

Wrap existing observability tools with simple query APIs.

Insight #3: Iterate with Operational Feedback

Context is a Living System

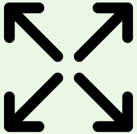
The best context schema emerges from real incident patterns.

Measure

Track which context fields engineers actually use during incidents.

Refine

Add missing context, remove unused fields.



Expand

Replicate and apply learnings to other services incrementally.

Insight #3: Iterate with Operational Feedback

Anti-Pattern To Avoid

Don't try to design the "perfect" context schema upfront. Let operational reality guide you.

What’s Next?

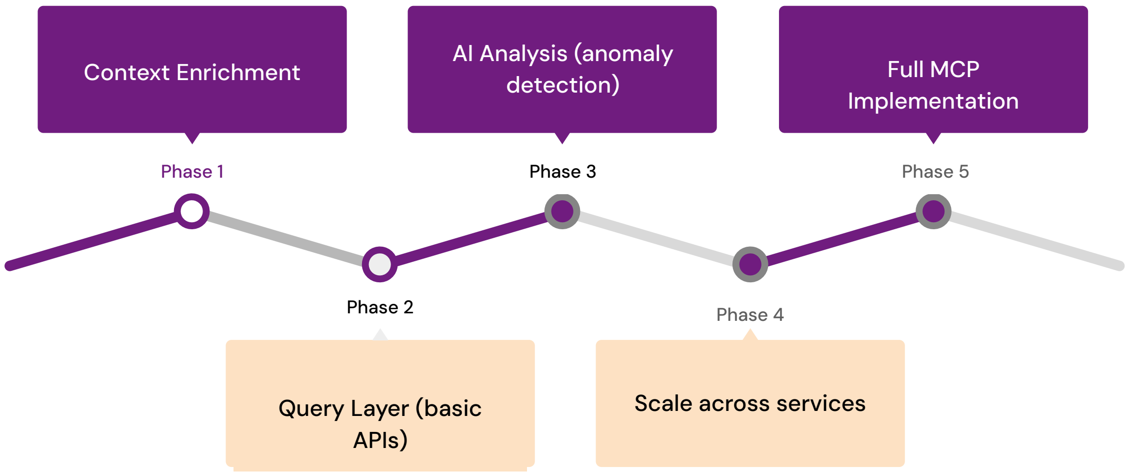
Industry Adoption & Scalability

Universal Applicability

Scales from startups to enterprises.

Ideal for Organizations With:

- Microservice Architectures
- Cloud-native deployments
- Distributed Systems at scale
- Non-uniform observability stack
- Alert fatigue problems
- Long MTTR



The Future of Observability

From Reactive to Proactive Approach

Traditional Observability

- Wait for alerts to happen
- Hunt for root-causes
- Manual Correlation
- Reactive firefighting
- Fatigued on-call engineers

AI-Powered Observability

- Predict issues before impact
- AI-suggested root causes
- Automatic correlation
- Proactive optimization
- Empowered on-call engineers

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

Thank You!

Pronnoy Goswami - Software Engineer @ Workday
pronnoy.goswami@gmail.com

linkedin.com/in/pronnoygoswami/

x.com/pronnoygoswami

github.com/goswamipronnoy