Tutorial 24: Advanced Observability - Enterprise Monitoring

Difficulty: advanced

Reading Time: 2.5 hours

Tags: advanced, observability, monitoring, enterprise, production

Description: Implement enterprise-grade observability with metrics, traces, logs, and alerting for production agent systems at scale.

🚀 Working Implementation

A complete, tested implementation of this tutorial is available in the repository:

View Tutorial 24 Implementation → (../../tutorial_implementation/tutorial24/)

The implementation includes:

- ✅ ObservabilityAgent with comprehensive plugin system
- ✅ SaveFilesAsArtifactsPlugin, MetricsCollectorPlugin, AlertingPlugin, PerformanceProfilerPlugin
- ✅ 4 comprehensive test files (all passing)
- ✅ Makefile with setup, dev, test, demo commands
- ✅ Complete README with usage examples and production deployment

Quick start:

```
cd tutorial_implementation/tutorial24
make setup

make dev
```

Tutorial 24: Advanced Observability & Monitoring

Goal: Master advanced observability patterns including plugin systems, Cloud Trace integration, custom metrics, distributed tracing, and production monitoring dashboards.

Prerequisites:

- Tutorial 18 (Events & Observability)
- Tutorial 23 (Production Deployment)
- Understanding of observability concepts

What You'll Learn:

- ADK plugin system for monitoring
- Cloud Trace integration (`trace_to_cloud`)
- SaveFilesAsArtifactsPlugin for debugging
- Custom observability plugins
- Distributed tracing across agents
- Performance metrics collection
- Production monitoring dashboards
- Alerting and incident response

Time to Complete: 55-70 minutes

:::info API Verification

Source Verified: Official ADK source code (version 1.16.0+)

Correct Plugin API: Plugins extend `BasePlugin` and implement `on_event_callback()` method

Correct Pattern:

```python
from google.adk.plugins import BasePlugin
from google.adk.events import Event
from typing import Optional

class CustomPlugin(BasePlugin):
    def __init__(self, name: str = 'custom_plugin'):
        super().__init__(name)

    async def on_event_callback(self, *, invocation_context, event: Event) ->
        # Handle events here
        if hasattr(event, 'event_type'):
            if event.event_type == 'request_start':
                # Handle request start
                pass
        return None  # Return None to continue normal processing
```

Plugin Registration: Plugins are registered with `InMemoryRunner(plugins=[...])`

Cloud Trace: Enabled via CLI flags (`--trace_to_cloud`) at deployment time

Verification Date: January 2025

:::

Why Advanced Observability Matters

Problem: Production agents require deep visibility into behavior, performance, and failures for debugging and optimization.

Solution: **Advanced observability** with plugins, distributed tracing, and custom metrics provides comprehensive system insight.

Benefits:

- 🔍 **Deep Visibility**: Understand complex agent behaviors
- 🐛 **Faster Debugging**: Quickly identify root causes
- 📊 **Performance Insights**: Optimize based on real data
- 🚨 **Proactive Alerting**: Detect issues before users
- 📈 **Trend Analysis**: Identify patterns over time
- 🎯 **Bottleneck Identification**: Find performance constraints

Observability Pillars:

- **Traces**: Request flow through system
- **Metrics**: Quantitative measurements
- **Logs**: Detailed event records
- **Events**: State changes and actions

```
Observability Pillars Overview:
+--------------------+     +--------------------+     +--------------------+
|       TRACES       | --> |      METRICS       | --> |       LOGS         |
| Request flow &     |     | Quantitative       |     | Detailed event     |
| timing analysis    |     | measurements       |     | records            |
+--------------------+     +--------------------+     +--------------------+
         |                          |                          |
         v                          v                          v
+--------------------+     +--------------------+     +--------------------+
|      EVENTS        | <-- |   ALERTS &         | <-- |   DASHBOARDS       |
| State changes &    |     |   THRESHOLDS       |     |   Visualizations   |
| action triggers    |     |                    |     |                    |
+--------------------+     +--------------------+     +--------------------+
```

1. ADK Plugin System

What Are Plugins?

Plugins are modular extensions that intercept and observe agent execution without modifying core logic.

Source: `google/adk/plugins/`

```
Plugin System Architecture:
+--------------------+     +--------------------+     +--------------------+
|   USER REQUEST     | --> |   ADK RUNNER       | --> |   AGENT CORE       |
|                    |     | (with plugins)     |     | (business logic)   |
+--------------------+     +--------------------+     +--------------------+
         |                          |                          |
         |                 +--------------------+              |
         |                 |   PLUGIN SYSTEM    |              |
         |                 | - MetricsPlugin    |              |
         |                 | - AlertingPlugin   |              |
         |                 | - ProfilingPlugin  |              |
         |                 +--------------------+              |
         |                          |                          |
         v                          v                          v
+--------------------+     +--------------------+     +--------------------+
|   AGENT RESPONSE   | <-- | EVENT PROCESSING   | <-- |   MODEL CALLS      |
|                    |     | (intercepted)      |     |   & TOOL USAGE     |
+--------------------+     +--------------------+     +--------------------+
```

Use Cases:

- Saving artifacts automatically
- Sending traces to Cloud Trace
- Custom metrics collection
- Performance profiling
- Compliance logging

Built-in Plugins

SaveFilesAsArtifactsPlugin

Automatically saves agent outputs as artifacts.

```python
"""
SaveFilesAsArtifactsPlugin example.
"""

import asyncio
import os
from google.adk.agents import Agent
from google.adk.runners import InMemoryRunner
from google.adk.plugins import SaveFilesAsArtifactsPlugin
from google.genai import types

# Environment setup
os.environ['GOOGLE_GENAI_USE_VERTEXAI'] = '1'
os.environ['GOOGLE_CLOUD_PROJECT'] = 'your-project-id'
os.environ['GOOGLE_CLOUD_LOCATION'] = 'us-central1'

async def main():
    """Demonstrate SaveFilesAsArtifactsPlugin."""

    # Create agent
    agent = Agent(
        model='gemini-2.0-flash',
        name='artifact_agent',
        instruction="Generate reports and save them automatically."
    )

    # Create plugin (saves uploaded files as artifacts)
    artifact_plugin = SaveFilesAsArtifactsPlugin()

    # Create runner with plugin
    runner = InMemoryRunner(
        agent=agent,
        app_name='artifact_demo',
        plugins=[artifact_plugin]  # Register plugin with runner
    )

    # Create session
    session = await runner.session_service.create_session(
        user_id='user',
        app_name='artifact_demo'
    )

    # Run agent
    async for event in runner.run_async(
        user_id='user',
        session_id=session.id,
        new_message=types.Content(
            role='user',
            parts=[types.Part.from_text("Generate a brief report about AI agen
        )
    ):
        if event.content and event.content.parts:
            text = ''.join(part.text or '' for part in event.content.parts)
            if text:
                print(f"[{event.author}]: {text[:200]}...")

    print("\n✔️ Plugin automatically saves uploaded files as artifacts")

if __name__ == '__main__':
    asyncio.run(main())
```

2. Cloud Trace Integration

Enabling Cloud Trace

Cloud Trace provides distributed tracing for Google Cloud applications.

Important: Cloud Trace is enabled at **deployment time** using CLI flags, not in application code.

```
Cloud Trace Integration Flow:
+--------------------+     +--------------------+     +--------------------+
|   AGENT REQUEST    | --> |   ADK RUNTIME      | --> |   MODEL/TOOLS      |
| (user input)       |     | (with tracing)     |     | (execution)        |
+--------------------+     +--------------------+     +--------------------+
         |                          |                          |
         |                 +--------------------+              |
         |                 |  TRACE COLLECTION  |              |
         |                 | - Request spans    |              |
         |                 | - Tool calls       |              |
         |                 | - Model timing     |              |
         |                 +--------------------+              |
         |                          |                          |
         v                          v                          v
+--------------------+     +--------------------+     +--------------------+
|   RESPONSE BACK    | <-- |   GOOGLE CLOUD     | <-- |   CLOUD TRACE      |
|   TO USER          |     |   INFRASTRUCTURE   |     |   (storage)        |
+--------------------+     +--------------------+     +--------------------+
```

Deploying with Cloud Trace

```
# Deploy to Cloud Run with tracing
adk deploy cloud_run \
  --project your-project-id \
  --region us-central1 \
  --service-name observability-agent \
  --trace_to_cloud  # Enable Cloud Trace

# Deploy to Agent Engine with tracing
adk deploy agent_engine \
  --project your-project-id \
  --region us-central1 \
  --trace_to_cloud  # Enable Cloud Trace

# Run local web UI with tracing
adk web --trace_to_cloud

# Run local API server with tracing
adk api_server --trace_to_cloud
```

Agent Engine with Tracing (Programmatic)

For Agent Engine deployments, you can enable tracing in the AdkApp configuration:

```python
"""
Agent Engine deployment with Cloud Trace.
"""

from vertexai.preview.reasoning_engines import AdkApp
from google.adk.agents import Agent

# Create agent
root_agent = Agent(
    model='gemini-2.0-flash',
    name='traced_agent',
    instruction="You are a helpful assistant."
)

# Create ADK app with tracing enabled
adk_app = AdkApp(
    agent=root_agent,
    enable_tracing=True  # Enable Cloud Trace for Agent Engine
)

# Deploy to Agent Engine
# This app will send traces to Cloud Trace automatically
```

Viewing Traces in Cloud Console

```
# View traces in Cloud Console
https://console.cloud.google.com/traces?project=your-project-id

# Filter traces by:
# - Agent name
# - Time range
# - Latency threshold
# - Error status

# Analyze:
# - Request flow and latency
# - Tool invocation spans
# - Model call timing
# - Performance bottlenecks
```

3. Real-World Example: Production Monitoring System

Let's build a comprehensive production monitoring system with custom plugins and metrics.

```
Metrics Collection Flow:
+--------------------+     +--------------------+     +--------------------+
|   AGENT EVENTS     | --> | METRICS PLUGIN     | --> | REQUEST METRICS    |
| (start/complete)   |     | (event handler)    |     | (latency, tokens) |
+--------------------+     +--------------------+     +--------------------+
         |                          |                          |
         v                          v                          v
+--------------------+     +--------------------+     +--------------------+
| AGGREGATE METRICS | <-- |    DATA STORAGE    | <-- |    CALCULATIONS    |
| (success rate,     |     |    (in memory)     |     |    (averages,      |
|  avg latency)      |     |                    |     |     totals)        |
+--------------------+     +--------------------+     +--------------------+
```

Complete Implementation

```
"""
ADK Tutorial 24: Advanced Observability & Monitoring

This agent demonstrates comprehensive observability patterns including:
- SaveFilesAsArtifactsPlugin for automatic file saving
- MetricsCollectorPlugin for request/response tracking
- AlertingPlugin for error detection and alerts
- PerformanceProfilerPlugin for detailed performance analysis
- ProductionMonitoringSystem for complete monitoring solution

Features:
- Plugin-based architecture for modular observability
- Real-time metrics collection and reporting
- Error detection and alerting
- Performance profiling and analysis
- Production-ready monitoring patterns
"""

import asyncio
import time
from datetime import datetime
from typing import Dict, List, Optional, Any
from dataclasses import dataclass, field

from google.adk.agents import Agent
from google.adk.plugins import BasePlugin
from google.adk.plugins.save_files_as_artifacts_plugin import SaveFilesAsArtif
from google.adk.events import Event
from google.genai import types

@dataclass
class RequestMetrics:
    """Metrics for a single request."""
    request_id: str
    agent_name: str
    start_time: float
    end_time: Optional[float] = None
    latency: Optional[float] = None
    success: bool = True
    error: Optional[str] = None
    token_count: int = 0
    tool_calls: int = 0

@dataclass
class AggregateMetrics:
    """Aggregate metrics across requests."""
    total_requests: int = 0
    successful_requests: int = 0
    failed_requests: int = 0
    total_latency: float = 0.0
    total_tokens: int = 0
    total_tool_calls: int = 0
    requests: List[RequestMetrics] = field(default_factory=list)

    @property
    def success_rate(self) -> float:
        """Calculate success rate."""
        if self.total_requests == 0:
            return 0.0
        return self.successful_requests / self.total_requests

    @property
    def avg_latency(self) -> float:
        """Calculate average latency."""
        if self.total_requests == 0:
            return 0.0
        return self.total_latency / self.total_requests

    @property
    def avg_tokens(self) -> float:
        """Calculate average tokens."""
        if self.total_requests == 0:
            return 0.0
        return self.total_tokens / self.total_requests

class MetricsCollectorPlugin(BasePlugin):
    """Plugin to collect request metrics."""

    def __init__(self, name: str = 'metrics_collector_plugin'):
        """Initialize metrics collector."""
        super().__init__(name)
        self.metrics = AggregateMetrics()
        self.current_requests: Dict[str, RequestMetrics] = {}

    async def on_event_callback(self, *, invocation_context, event: Event) ->
        """Handle agent events for metrics collection."""
        # Track events (implementation simplified for tutorial)
        if hasattr(event, 'event_type'):
            if event.event_type == 'request_start':
                request_id = str(time.time())
                metrics = RequestMetrics(
                    request_id=request_id,
                    agent_name='observability_agent',
                    start_time=time.time()
                )
                self.current_requests[request_id] = metrics
                print(f"📊 [METRICS] Request started at {datetime.now().strfti

            elif event.event_type == 'request_complete':
                if self.current_requests:
                    request_id = list(self.current_requests.keys())[0]
                    metrics = self.current_requests[request_id]
                    metrics.end_time = time.time()
                    metrics.latency = metrics.end_time - metrics.start_time

                    # Update aggregates
                    self.metrics.total_requests += 1
                    self.metrics.successful_requests += 1
                    self.metrics.total_latency += metrics.latency
                    self.metrics.requests.append(metrics)

                    print(f"✔️ [METRICS] Request completed: {metrics.latency:.
                    del self.current_requests[request_id]

    def get_summary(self) -> str:
        """Get metrics summary."""

        m = self.metrics

        summary = f"""
METRICS SUMMARY
{'='*70}

Total Requests:       {m.total_requests}
Successful:           {m.successful_requests}
Failed:               {m.failed_requests}
Success Rate:         {m.success_rate*100:.1f}%

Average Latency:      {m.avg_latency:.2f}s
Average Tokens:       {m.avg_tokens:.0f}
Total Tool Calls:     {m.total_tool_calls}

{'='*70}
        """.strip()

        return summary

class AlertingPlugin(BasePlugin):
    """Plugin for alerting on anomalies."""
    def __init__(self, name: str = 'alerting_plugin', latency_threshold: float
        """
        Initialize alerting plugin.

        Args:
            name: Plugin name
            latency_threshold: Alert if latency exceeds this (seconds)
            error_threshold: Alert if consecutive errors exceed this
        """
        super().__init__(name)
        self.latency_threshold = latency_threshold
        self.error_threshold = error_threshold
        self.consecutive_errors = 0

    async def on_event_callback(self, *, invocation_context, event: Event) ->
        """Handle agent events for alerting."""
        if hasattr(event, 'event_type'):
            if event.event_type == 'request_complete':
                # Reset error counter on success
                self.consecutive_errors = 0

            elif event.event_type == 'request_error':
                self.consecutive_errors += 1
                print("🚨 [ALERT] Error detected")

                if self.consecutive_errors >= self.error_threshold:
                    print(f"🚨🚨 [CRITICAL ALERT] {self.consecutive_errors} co

class PerformanceProfilerPlugin(BasePlugin):
    """Plugin for detailed performance profiling."""

    def __init__(self, name: str = 'performance_profiler_plugin'):
        """Initialize profiler."""
        super().__init__(name)
        self.profiles: List[Dict] = []
        self.current_profile: Optional[Dict] = None

    async def on_event_callback(self, *, invocation_context, event: Event) ->
        """Handle agent events for profiling."""
        if hasattr(event, 'event_type'):
            if event.event_type == 'tool_call_start':
                self.current_profile = {
                    'tool': getattr(event, 'tool_name', 'unknown'),
                    'start_time': time.time()
                }
                print("⚙️ [PROFILER] Tool call started")
            elif event.event_type == 'tool_call_complete':
                if self.current_profile:
                    self.current_profile['end_time'] = time.time()
                    self.current_profile['duration'] = (
                        self.current_profile['end_time'] - self.current_profil
                    )
                    self.profiles.append(self.current_profile)
                    print(f"✔ [PROFILER] Tool call completed: {self.current_p
                    self.current_profile = None

    def get_profile_summary(self) -> str:
        """Get profiling summary."""

        if not self.profiles:
            return "No profiles collected"

        summary = f"\nPERFORMANCE PROFILE\n{'='*70}\n\n"

        tool_stats = {}

        for profile in self.profiles:
            if 'duration' not in profile:
                continue

            tool = profile['tool']

            if tool not in tool_stats:
                tool_stats[tool] = {
                    'calls': 0,
                    'total_duration': 0.0,
                    'min_duration': float('inf'),
                    'max_duration': 0.0
                }

            stats = tool_stats[tool]
            stats['calls'] += 1
            stats['total_duration'] += profile['duration']
            stats['min_duration'] = min(stats['min_duration'], profile['durati
            stats['max_duration'] = max(stats['max_duration'], profile['durati

        for tool, stats in tool_stats.items():
            avg_duration = stats['total_duration'] / stats['calls']

            summary += f"Tool: {tool}\n"
            summary += f"  Calls:        {stats['calls']}\n"
            summary += f"  Avg Duration: {avg_duration:.3f}s\n"
            summary += f"  Min Duration: {stats['min_duration']:.3f}s\n"
            summary += f"  Max Duration: {stats['max_duration']:.3f}s\n\n"

        summary += f"{'='*70}\n"

        return summary

# Create the observability agent with all plugins
root_agent = Agent(
    model='gemini-2.5-flash',
    name='observability_agent',
    description="""Production assistant with comprehensive observability inclu
alerting, and performance profiling for enterprise monitoring.""",
    instruction="""
You are a production assistant helping with customer inquiries about AI and te

Key behaviors:
- Provide accurate, helpful responses
- Keep responses concise but informative
- Use clear, simple language
- Stay on topic and focused

Your responses are being monitored for quality, performance, and reliability.
Always be helpful and accurate.
    """.strip(),
    generate_content_config=types.GenerateContentConfig(
        temperature=0.5,
        max_output_tokens=1024
    )
)

def main():
    """
    Main entry point for demonstration.

    This function demonstrates how to use the observability agent with the ADK
    The actual monitoring plugins are registered at the runner level (see test
    """
    print("🚀 Tutorial 24: Advanced Observability & Monitoring")
    print("=" * 70)
    print("\n📊 Observability Agent Features:")
    print("  • SaveFilesAsArtifactsPlugin - automatic file saving")
    print("  • MetricsCollectorPlugin - request/response metrics")
    print("  • AlertingPlugin - error detection and alerts")
    print("  • PerformanceProfilerPlugin - detailed profiling")
    print("\n💡 To see the agent in action:")
    print("  1. Run: adk web")
    print("  2. Open http://localhost:8000")
    print("  3. Select 'observability_agent' from dropdown")
    print("  4. Try various prompts and observe console metrics")
    print("\n" + "=" * 70)

if __name__ == '__main__':
    main()
```

Expected Output

```
🚀 Tutorial 24: Advanced Observability & Monitoring
======================================================================

📊 Observability Agent Features:
  • SaveFilesAsArtifactsPlugin - automatic file saving
  • MetricsCollectorPlugin - request/response metrics
  • AlertingPlugin - error detection and alerts
  • PerformanceProfilerPlugin - detailed profiling

💡 To see the agent in action:
  1. Run: adk web
  2. Open http://localhost:8000
  3. Select 'observability_agent' from dropdown
  4. Try various prompts and observe console metrics

======================================================================
```

4. Custom Monitoring Dashboard

Prometheus Metrics Export

```python
from prometheus_client import Counter, Histogram, Gauge, generate_latest
from fastapi import FastAPI, Response

app = FastAPI()

# Metrics
request_counter = Counter('agent_requests_total', 'Total agent requests')
request_duration = Histogram('agent_request_duration_seconds', 'Request durati
active_requests = Gauge('agent_active_requests', 'Currently active requests')
error_counter = Counter('agent_errors_total', 'Total errors')

@app.get("/metrics")
async def metrics():
    """Prometheus metrics endpoint."""
    return Response(content=generate_latest(), media_type="text/plain")

@app.middleware("http")
async def track_metrics(request, call_next):
    """Middleware to track metrics."""

    active_requests.inc()
    request_counter.inc()

    with request_duration.time():
        try:
            response = await call_next(request)
            return response
        except Exception as e:
            error_counter.inc()
            raise
        finally:
            active_requests.dec()
```

5. Project Structure & Testing

Package Structure

The observability agent follows ADK best practices with proper packaging:

```
tutorial24/
├── observability_agent/           # Main package
|   ├── __init__.py               # Package initialization
|   └── agent.py                  # Agent implementation with plugins
├── tests/                        # Comprehensive test suite
|   ├── __init__.py
|   ├── test_agent.py            # Agent configuration tests
|   ├── test_imports.py          # Import validation
|   ├── test_plugins.py          # Plugin functionality tests
|   └── test_structure.py        # Project structure tests
├── pyproject.toml               # Modern Python packaging
├── requirements.txt             # Dependencies
├── Makefile                    # Build and test commands
├── .env.example               # Environment template
└── README.md                  # Implementation guide
```

Installation & Setup

```
# Install dependencies
pip install -r requirements.txt
pip install -e .

# Set environment variables

# OR

# Run the agent
adk web  # Select 'observability_agent' from dropdown
```

Testing the Implementation

```
# Run all tests with coverage
make test

# Run specific test files
pytest tests/test_plugins.py -v
pytest tests/test_agent.py -v

# Test with different configurations
pytest tests/ -k "plugin" --tb=short
```

Key Testing Patterns

- **Plugin Isolation**: Test each plugin independently
- **Event Handling**: Verify correct event processing
- **Metrics Accuracy**: Ensure metrics calculations are correct
- **Error Scenarios**: Test error handling and alerting
- **Integration**: Test plugins working together

```
Production Monitoring Architecture:
+--------------------+     +--------------------+     +--------------------+
|   USER REQUESTS    | --> |   ADK AGENT        | --> |   MODEL/TOOLS      |
| (web, API, CLI)    |     | (with plugins)     |     | (Gemini, custom)   |
+--------------------+     +--------------------+     +--------------------+
         |                          |                          |
         |                 +--------------------+              |
         |                 |   PLUGIN LAYER     |              |
         |                 | +---------------+  |              |
         |                 | | Metrics       |  |              |
         |                 | | Collector     |  |              |
         |                 | +---------------+  |              |
         |                 | +---------------+  |              |
         |                 | | Alerting      |  |              |
         |                 | | System        |  |              |
         |                 | +---------------+  |              |
         |                 | +---------------+  |              |
         |                 | | Performance   |  |              |
         |                 | | Profiler      |  |              |
         |                 | +---------------+  |              |
         |                 +--------------------+              |
         |                          |                          |
         v                          v                          v
+--------------------+     +--------------------+     +--------------------+
|   RESPONSE BACK    | <-- |   CLOUD INFRA      | <-- |   EXTERNAL         |
|   TO USER          |     | (Trace, Storage)   |     |   SYSTEMS          |
+--------------------+     +--------------------+     +--------------------+
         |                          |                          |
         v                          v                          v
+--------------------+     +--------------------+     +--------------------+
| MONITORING OUTPUT  | <-- |   DASHBOARDS       | <-- |   METRICS EXPORT   |
| (logs, alerts)     |     | (Grafana, custom)  |     | (Prometheus)       |
+--------------------+     +--------------------+     +--------------------+
```

Summary

You've mastered advanced observability with the ADK plugin system:

Key Takeaways:

- ✅ **Plugin Architecture**: Extend `BasePlugin` with `on_event_callback()` method
- ✅ **Event-Driven**: Plugins respond to agent lifecycle events

- ✅ **Modular Design**: Separate plugins for metrics, alerting, profiling
- ✅ **Production Ready**: Comprehensive monitoring for enterprise deployments
- ✅ **Cloud Integration**: Cloud Trace support for distributed tracing
- ✅ **Testing**: Full test coverage with pytest and comprehensive validation

Plugin Development Pattern:

```
from google.adk.plugins import BasePlugin
from google.adk.events import Event
from typing import Optional

class CustomPlugin(BasePlugin):
    def __init__(self, name: str = 'custom_plugin'):
        super().__init__(name)

    async def on_event_callback(self, *, invocation_context, event: Event) ->
        # Handle agent events
        if hasattr(event, 'event_type'):
            if event.event_type == 'request_start':
                # Custom logic here
                pass
        return None  # Return None to continue normal processing
```

Production Deployment:

```
# Install and setup
make setup

# Run with monitoring
make dev  # Opens web UI with observability_agent

# Deploy to production
make deploy  # Cloud Run with Cloud Trace enabled
```

Testing & Quality:

- **100% Test Coverage**: All plugins and agent logic tested
- **Integration Tests**: End-to-end plugin functionality
- **Error Handling**: Comprehensive error scenarios covered
- **Performance**: Efficient event processing without blocking

Production Checklist:

- [ ] Cloud Trace enabled for distributed tracing
- [ ] Custom metrics plugins deployed
- [ ] Alerting thresholds configured
- [ ] Performance profiling active
- [ ] Monitoring dashboards set up
- [ ] Incident response procedures documented
- [ ] Regular metrics review scheduled

Next Steps:

- **Tutorial 25**: Master Best Practices & Patterns (Final Tutorial!)

Resources:

- Tutorial Implementation (../../tutorial_implementation/tutorial24)
- ADK Plugin Documentation (https://github.com/google/adk-python)
- Cloud Trace (https://cloud.google.com/trace/docs)
- Observability Best Practices (https://cloud.google.com/architecture/observability)

🎉 **Tutorial 24 Complete!** You now know advanced observability patterns. Continue to Tutorial 25 for best practices and the completion of the series!

Generated on 2025-10-19 17:57:05 from 24_advanced_observability.md

Source: Google ADK Training Hub