

# Tutorial 18: Events and Observability - Agent Monitoring

---

**Difficulty:** advanced

**Reading Time:** 2 hours

**Tags:** advanced, observability, monitoring, events, metrics

**Description:** Implement comprehensive observability for agents with event tracking, metrics collection, and monitoring dashboards for production systems.



## Working Implementation

---

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

[View Tutorial 18 Implementation](#) → ([../.. /tutorial\\_implementation/tutorial18/](#))

The implementation includes:

- ✓ CustomerServiceMonitor with comprehensive event tracking
- ✓ EventLogger, MetricsCollector, and EventAlerter classes
- ✓ 49 comprehensive tests (all passing)
- ✓ Makefile with setup, dev, test, demo commands
- ✓ Complete README with usage examples

Quick start:

```
cd tutorial_implementation/tutorial18
make setup

make dev
```

# Tutorial 18: Events & Observability

---

**Goal:** Master event tracking and observability patterns to monitor agent behavior, debug issues, and gain insights into agent decision-making processes in production systems.

**Prerequisites:**

- Tutorial 01 (Hello World Agent)
- Tutorial 06 (Multi-Agent Systems)
- Tutorial 09 (Callbacks & Guardrails)
- Understanding of logging and monitoring concepts

**What You'll Learn:**

- Understanding `Event` class and event lifecycle
- Using `EventActions` for state changes and agent transfers
- Implementing observability with trace views
- Tracking long-running tool operations
- Building monitoring dashboards
- Debugging agent behavior with event logs
- Best practices for production observability

**Time to Complete:** 55-70 minutes

---






## Why Events & Observability Matter

---

**Problem:** Without visibility into agent execution, debugging failures and understanding agent behavior is difficult.

**Solution:** **Events** provide structured logs of agent activity, while **observability** tools make these events actionable.

**Benefits:**

-  **Visibility:** See exactly what agents are doing
-  **Debugging:** Identify failures and bottlenecks
-  **Analytics:** Track performance metrics
- [FLOW] **State Tracking:** Monitor state changes over time
-  **Optimization:** Find inefficiencies
-  **Alerting:** Detect anomalies in real-time

**What Events Capture:**

- Agent invocations
  - Tool calls
  - State modifications
  - Agent transfers
  - Errors and exceptions
  - Timing information
  - Authentication requests
- 

# 1. Event System Basics

---

## | What is an Event?

An **Event** extends `LlmResponse` and represents a discrete action or state change during agent execution.

**Source:** `google/adk/events/event.py`

**Event Structure:**

```
from google.adk.events import Event, EventActions
from google.genai import types

event = Event(
    invocation_id='inv-123',          # Unique invocation identifier
    author='agent_name',              # Agent that created event
    content=types.Content(...),      # Event content/message
    actions=EventActions(            # Actions to perform
        state_delta={'key': 'value'}, # State changes
        artifact_delta={'file': 1},   # Artifact changes
        escalate=False,               # Escalate to human
        transfer_to_agent='other_agent' # Transfer to another agent
    )
)
```

## Event Lifecycle

1. **Creation** → Agent generates event
2. **Processing** → ADK processes event actions
3. **State Update** → State modified per `state_delta`
4. **Logging** → Event logged for observability
5. **Next Action** → Based on event actions (continue, transfer, escalate)

## 2. EventActions: Controlling Agent Flow

### State Delta

Modify session state:

```
from google.adk.events import EventActions

# Create event with state changes
actions = EventActions(
    state_delta={
        'user_preference': 'dark_mode',
        'last_query_time': '2025-10-08T14:30:00Z',
        'query_count': 5
    }
)

# State will be updated in session
# Accessible in subsequent agent calls
```

## Artifact Delta

Track artifact changes:

```
actions = EventActions(
    artifact_delta={
        'report.pdf': 1,      # Version 1 of report.pdf created
        'data.csv': 2,       # Version 2 of data.csv created
        'image.png': 1       # Version 1 of image.png created
    }
)

# Tracks which artifacts were created/modified
# Useful for auditing and provenance
```

## Agent Transfer

Transfer control to another agent:

```
actions = EventActions(  
    transfer_to_agent='specialized_agent',  
    state_delta={'transfer_reason': 'requires_expertise'}  
)  
  
# Current agent stops  
# Control transfers to 'specialized_agent'  
# State preserved and passed along
```

## Escalation

Escalate to human review:

```
actions = EventActions(  
    escalate=True,  
    state_delta={'escalation_reason': 'ambiguous_request'}  
)  
  
# Agent pauses  
# Human review requested  
# Common for: errors, sensitive operations, low confidence
```

## Skip Summarization

Control whether event is summarized:

```
actions = EventActions(  
    skip_summarization=True  
)  
  
# Event won't be included in context summaries  
# Useful for: verbose logs, interim states, debug info
```

## Long-Running Tools

Track asynchronous operations:

```
actions = EventActions(  
    long_running_tool_ids=['tool_async_123', 'tool_async_456']  
)  
  
# Marks tools as running asynchronously  
# Allows agent to continue while tools execute  
# Results integrated when available
```

---

## 3. Real-World Example: Customer Service with Event Tracking

---

Let's build a customer service system with comprehensive event tracking and observability.

# | Complete Implementation



```

"""
Customer Service Agent with Event Tracking
Monitors all agent actions, state changes, and escalations.
"""

import asyncio
import os
from datetime import datetime
from typing import List, Dict
from google.adk.agents import Agent, Runner, Session
from google.adk.events import Event, EventActions
from google.adk.tools import FunctionTool
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'

class CustomerServiceMonitor:
    """Customer service with comprehensive event monitoring."""

    def __init__(self):
        """Initialize customer service system."""

        # Event log storage
        self.events: List[Dict] = []

        # Create tools with event tracking

    def check_order_status(order_id: str) -> str:
        """Check order status."""
        self._log_tool_call('check_order_status', {'order_id': order_id})

        # Simulated order lookup
        status = {
            'ORD-001': 'shipped',
            'ORD-002': 'processing',
            'ORD-003': 'delivered'
        }.get(order_id, 'not_found')

        return f"Order {order_id} status: {status}"

    def process_refund(order_id: str, amount: float) -> str:
        """Process refund request."""
        self._log_tool_call('process_refund', {

```

```

        'order_id': order_id,
        'amount': amount
    })

    # This would trigger escalation for amounts > 100
    if amount > 100:
        return "ESCALATE: Refund exceeds approval threshold"

    return f"Refund of ${amount} approved for order {order_id}"

def check_inventory(product_id: str) -> str:
    """Check product inventory."""
    self._log_tool_call('check_inventory', {'product_id': product_id})

    # Simulated inventory check
    inventory = {
        'PROD-A': 150,
        'PROD-B': 5,
        'PROD-C': 0
    }.get(product_id, 0)

    return f"Product {product_id} inventory: {inventory} units"

# Customer service agent
self.agent = Agent(
    model='gemini-2.0-flash',
    name='customer_service',
    description='Customer service agent with event tracking',
    instruction="""

```

You are a customer service agent helping customers with:

- Order status inquiries
- Refund requests
- Inventory checks
- General questions

Guidelines:

1. Always be polite and helpful
2. Use tools to get accurate information
3. For refunds > \$100, escalate to supervisor
4. Track all interactions in state
5. Log important decisions

Tools available:

- check\_order\_status: Get order status
  - process\_refund: Process refund (escalate if > \$100)
  - check\_inventory: Check product availability
- ```

    """ .strip(),

```

```

        tools=[
            FunctionTool(check_order_status),
            FunctionTool(process_refund),
            FunctionTool(check_inventory)
        ],
        generate_content_config=types.GenerateContentConfig(
            temperature=0.5,
            max_output_tokens=1024
        )
    )

    self.runner = Runner()

def _log_tool_call(self, tool_name: str, args: Dict):
    """Log tool invocation."""
    self.events.append({
        'timestamp': datetime.now().isoformat(),
        'type': 'tool_call',
        'tool': tool_name,
        'arguments': args
    })

def _log_agent_event(self, event_type: str, data: Dict):
    """Log agent event."""
    self.events.append({
        'timestamp': datetime.now().isoformat(),
        'type': event_type,
        'data': data
    })

async def handle_customer_query(self, customer_id: str, query: str):
    """
    Handle customer query with full event tracking.

    Args:
        customer_id: Customer identifier
        query: Customer query
    """

    print(f"\n{'='*70}")
    print(f"CUSTOMER: {customer_id}")
    print(f"QUERY: {query}")
    print(f"\n{'='*70}")

    # Log query event
    self._log_agent_event('customer_query', {
        'customer_id': customer_id,

```

```

        'query': query
    })

    # Create session with customer context
    session = Session()
    session.state['customer_id'] = customer_id
    session.state['query_time'] = datetime.now().isoformat()
    session.state['query_count'] = session.state.get('query_count', 0) + 1

    # Execute agent
    result = await self.runner.run_async(
        query,
        agent=self.agent,
        session=session
    )

    # Log response
    response_text = result.content.parts[0].text

    self._log_agent_event('agent_response', {
        'customer_id': customer_id,
        'response': response_text
    })

    # Check for escalation
    if 'ESCALATE' in response_text:
        self._log_agent_event('escalation', {
            'customer_id': customer_id,
            'reason': response_text
        })
        print("🚨 ESCALATED TO SUPERVISOR\n")

    print(f"🤖 AGENT RESPONSE:\n{response_text}\n")
    print(f"{'='*70}\n")

    return result

def get_event_summary(self) -> str:
    """Generate event summary report."""

    total_events = len(self.events)

    event_types = {}
    for event in self.events:
        event_type = event['type']
        event_types[event_type] = event_types.get(event_type, 0) + 1

```

```

    tool_calls = [e for e in self.events if e['type'] == 'tool_call']
    escalations = [e for e in self.events if e['type'] == 'escalation']

    summary = f"""
EVENT SUMMARY REPORT
{'='*70}

Total Events: {total_events}

Event Types:
"""

    for event_type, count in event_types.items():
        summary += f"    - {event_type}: {count}\n"

    summary += f"\nTool Calls: {len(tool_calls)}\n"

    if tool_calls:
        summary += "    Tools Used:\n"
        tool_usage = {}
        for call in tool_calls:
            tool = call['tool']
            tool_usage[tool] = tool_usage.get(tool, 0) + 1

        for tool, count in tool_usage.items():
            summary += f"        - {tool}: {count} calls\n"

    summary += f"\nEscalations: {len(escalations)}\n"

    if escalations:
        summary += "    Escalation Reasons:\n"
        for esc in escalations:
            summary += f"        - {esc['data']['reason']}\n"

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

    return summary

def get_detailed_timeline(self) -> str:
    """Get detailed event timeline."""

    timeline = f"\nDETAILED EVENT TIMELINE\n{'='*70}\n"

    for i, event in enumerate(self.events, 1):
        timeline += f"\n[{i}] {event['timestamp']}\n"
        timeline += f"    Type: {event['type']}\n"

```

```

        if event['type'] == 'tool_call':
            timeline += f"    Tool: {event['tool']}\n"
            timeline += f"    Args: {event['arguments']}\n"
        elif event['type'] in ['customer_query', 'agent_response', 'escalation']:
            for key, value in event['data'].items():
                timeline += f"    {key}: {value}\n"

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

    return timeline

async def main():
    """Main entry point."""

    monitor = CustomerServiceMonitor()

    # Customer 1: Order status inquiry
    await monitor.handle_customer_query(
        customer_id='CUST-001',
        query='What is the status of my order ORD-001?'
    )

    await asyncio.sleep(1)

    # Customer 2: Refund request (small amount)
    await monitor.handle_customer_query(
        customer_id='CUST-002',
        query='I want a refund of $50 for order ORD-002'
    )

    await asyncio.sleep(1)

    # Customer 3: Refund request (large amount - triggers escalation)
    await monitor.handle_customer_query(
        customer_id='CUST-003',
        query='I need a refund of $150 for order ORD-003'
    )

    await asyncio.sleep(1)

    # Customer 4: Inventory check
    await monitor.handle_customer_query(
        customer_id='CUST-004',
        query='Is product PROD-B in stock?'
    )

    # Generate reports

```

```
print("\n" + monitor.get_event_summary())
print(monitor.get_detailed_timeline())

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

# | Expected Output



=====

CUSTOMER: CUST-001

QUERY: What is the status of my order ORD-001?

=====

🤖 AGENT RESPONSE:

Your order ORD-001 has been shipped! You should receive it soon.

=====

=====

CUSTOMER: CUST-002

QUERY: I want a refund of \$50 for order ORD-002

=====

🤖 AGENT RESPONSE:

I've processed your refund of \$50 for order ORD-002. The funds should appear in your account within 3-5 business days.

=====

=====

CUSTOMER: CUST-003

QUERY: I need a refund of \$150 for order ORD-003

=====

🚨 ESCALATED TO SUPERVISOR

🤖 AGENT RESPONSE:

ESCALATE: Refund exceeds approval threshold. This request requires supervisor approval. A supervisor will contact you within 24 hours to process your \$150 refund for order ORD-003.

=====

=====

CUSTOMER: CUST-004

QUERY: Is product PROD-B in stock?

=====

🤖 AGENT RESPONSE:

Product PROD-B currently has 5 units in stock. It's available for purchase, but inventory is running low. I recommend ordering soon if you're interested!

=====

## EVENT SUMMARY REPORT

---

Total Events: 12

Event Types:

- customer\_query: 4
- tool\_call: 4
- agent\_response: 4

Tool Calls: 4

Tools Used:

- check\_order\_status: 1 calls
- process\_refund: 2 calls
- check\_inventory: 1 calls

Escalations: 1

Escalation Reasons:

- ESCALATE: Refund exceeds approval threshold

---

## DETAILED EVENT TIMELINE

---

[1] 2025-10-08T14:30:15.123456

Type: customer\_query

customer\_id: CUST-001

query: What is the status of my order ORD-001?

[2] 2025-10-08T14:30:15.234567

Type: tool\_call

Tool: check\_order\_status

Args: {'order\_id': 'ORD-001'}

[3] 2025-10-08T14:30:16.345678

Type: agent\_response

customer\_id: CUST-001

response: Your order ORD-001 has been shipped! You should receive it soon.

[4] 2025-10-08T14:30:17.456789

Type: customer\_query

customer\_id: CUST-002

query: I want a refund of \$50 for order ORD-002

[5] 2025-10-08T14:30:17.567890

Type: tool\_call

```
Tool: process_refund
Args: {'order_id': 'ORD-002', 'amount': 50.0}

[6] 2025-10-08T14:30:18.678901
Type: agent_response
customer_id: CUST-002
response: I've processed your refund of $50...

[7] 2025-10-08T14:30:19.789012
Type: customer_query
customer_id: CUST-003
query: I need a refund of $150 for order ORD-003

[8] 2025-10-08T14:30:19.890123
Type: tool_call
Tool: process_refund
Args: {'order_id': 'ORD-003', 'amount': 150.0}

[9] 2025-10-08T14:30:20.901234
Type: escalation
customer_id: CUST-003
reason: ESCALATE: Refund exceeds approval threshold

[10] 2025-10-08T14:30:20.912345
Type: agent_response
customer_id: CUST-003
response: ESCALATE: Refund exceeds approval threshold...

[11] 2025-10-08T14:30:22.023456
Type: customer_query
customer_id: CUST-004
query: Is product PROD-B in stock?

[12] 2025-10-08T14:30:22.134567
Type: tool_call
Tool: check_inventory
Args: {'product_id': 'PROD-B'}
```

=====

---

## 4. ADK Web Trace View

---

ADK provides a built-in web UI for viewing traces and events.

### | Starting Trace View

```
# Start ADK web interface
adk web

# Open browser to:
# http://localhost:8080

# Navigate to "Trace" tab
# View all agent executions, events, and state changes
```

### | Trace View Features

#### **Event Tab:**

- All events in chronological order
- Event type filtering
- Event content viewing
- State delta visualization

#### **Request Tab:**

- Agent invocations
- Input messages
- Configuration used
- Session information

#### **Response Tab:**

- Agent responses
- Tool call results
- Timing information
- Token usage

**Graph Tab:**

- Visual workflow representation
  - Agent transitions
  - Sub-agent calls
  - Tool dependencies
- 

## 5. Advanced Observability Patterns

---

### | Pattern 1: Custom Event Logger

Create custom event logging:

```

import logging
from typing import List, Dict
from google.adk.events import Event

class EventLogger:
    """Custom event logger for structured logging."""

    def __init__(self):
        self.logger = logging.getLogger('agent_events')
        self.logger.setLevel(logging.INFO)

        # Configure handlers
        handler = logging.FileHandler('agent_events.log')
        handler.setFormatter(logging.Formatter(
            '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
        ))
        self.logger.addHandler(handler)

    def log_event(self, event: Event):
        """Log event with structured data."""
        self.logger.info({
            'invocation_id': event.invocation_id,
            'author': event.author,
            'content': event.content.parts[0].text if event.content else None,
            'actions': {
                'state_delta': event.actions.state_delta if event.actions else None,
                'escalate': event.actions.escalate if event.actions else None
            }
        })

# Usage
logger = EventLogger()

# In agent execution:
# logger.log_event(event)

```

## Pattern 2: Metrics Collection

Collect performance metrics:

```

from dataclasses import dataclass
from typing import Dict, List
import time

@dataclass
class AgentMetrics:
    """Agent performance metrics."""
    invocation_count: int = 0
    total_latency: float = 0.0
    tool_call_count: int = 0
    error_count: int = 0
    escalation_count: int = 0

class MetricsCollector:
    """Collect agent metrics for monitoring."""

    def __init__(self):
        self.metrics: Dict[str, AgentMetrics] = {}

    def track_invocation(self, agent_name: str, latency: float,
                        had_error: bool = False, escalated: bool = False):
        """Track agent invocation metrics."""

        if agent_name not in self.metrics:
            self.metrics[agent_name] = AgentMetrics()

        m = self.metrics[agent_name]
        m.invocation_count += 1
        m.total_latency += latency

        if had_error:
            m.error_count += 1
        if escalated:
            m.escalation_count += 1

    def get_summary(self, agent_name: str) -> Dict:
        """Get metrics summary for agent."""

        if agent_name not in self.metrics:
            return {}

        m = self.metrics[agent_name]

        return {
            'invocations': m.invocation_count,
            'avg_latency': m.total_latency / m.invocation_count if m.invocation_count > 0 else 0,
        }

```

```
        'error_rate': m.error_count / m.invocation_count if m.invocation_c
        'escalation_rate': m.escalation_count / m.invocation_count if m.in
    }

# Usage
collector = MetricsCollector()

start = time.time()
# ... run agent ...
latency = time.time() - start

collector.track_invocation('customer_service', latency, had_error=False, escal

print(collector.get_summary('customer_service'))
```

## | Pattern 3: Real-Time Alerting

Alert on specific event patterns:



```

from typing import Callable, List
from google.adk.events import Event

class EventAlerter:
    """Alert on specific event patterns."""

    def __init__(self):
        self.rules: List[tuple[Callable, Callable]] = []

    def add_rule(self, condition: Callable[[Event], bool],
                alert_fn: Callable[[Event], None]):
        """Add alerting rule."""
        self.rules.append((condition, alert_fn))

    def check_event(self, event: Event):
        """Check event against all rules."""
        for condition, alert_fn in self.rules:
            if condition(event):
                alert_fn(event)

# Usage
alerter = EventAlerter()

# Alert on escalations
alerter.add_rule(
    condition=lambda e: e.actions and e.actions.escalate,
    alert_fn=lambda e: print(f"🔴 ALERT: Escalation in {e.author}")
)

# Alert on errors
alerter.add_rule(
    condition=lambda e: 'error' in str(e.content).lower(),
    alert_fn=lambda e: print(f"🔴 ALERT: Error detected in {e.author}")
)

# Alert on high-value transactions
alerter.add_rule(
    condition=lambda e: e.actions and e.actions.state_delta
                        and e.actions.state_delta.get('transaction_amount', 0)
    alert_fn=lambda e: print(f"💰 ALERT: High-value transaction in {e.author}")
)

# Check events
# alerter.check_event(event)

```

## 6. Best Practices

### ✓ DO: Log Important State Changes

```
# ✓ Good - Track critical state
actions = EventActions(
    state_delta={
        'order_status': 'shipped',
        'shipping_carrier': 'UPS',
        'tracking_number': '1Z999AA10123456784',
        'updated_at': '2025-10-08T14:30:00Z'
    }
)

# ✗ Bad - No state tracking
# Agent modifies order but doesn't log
```

### ✓ DO: Use Escalation Appropriately

```
# ✓ Good - Escalate when necessary
if refund_amount > 100:
    actions = EventActions(
        escalate=True,
        state_delta={'escalation_reason': 'high_value_refund'}
    )

# ✗ Bad - No escalation for risky operations
# Process large refund without approval
```

## ✓ DO: Track Long-Running Operations

```
# ✓ Good - Mark async tools
actions = EventActions(
    long_running_tool_ids=['video_processing_123', 'report_generation_456']
)

# Agent continues while tools run
# Results integrated asynchronously

# ✗ Bad - Blocking on long operations
# Agent waits for lengthy tool execution
```

## ✓ DO: Include Context in Events

```
# ✓ Good - Rich context
event = Event(
    invocation_id='inv-123',
    author='customer_service',
    content=types.Content(
        parts=[types.Part.from_text('Processed refund')]
    ),
    actions=EventActions(
        state_delta={
            'action': 'refund_processed',
            'customer_id': 'CUST-123',
            'order_id': 'ORD-456',
            'amount': 50.00,
            'timestamp': '2025-10-08T14:30:00Z',
            'agent': 'customer_service'
        }
    )
)

# ✗ Bad - Minimal context
event = Event(
    invocation_id='inv-123',
    author='agent',
    content=types.Content(parts=[types.Part.from_text('Done')])
)
```

## 7. Troubleshooting

### | Issue: "Events not appearing in trace view"

#### Solutions:

##### 1. Ensure ADK web running:

```
adk web
# Check http://localhost:8080
```

##### 1. Verify logging enabled:

```
import logging
logging.basicConfig(level=logging.INFO)

# ADK will log events
```

##### 1. Check event structure:

```
# Events must have required fields
event = Event(
    invocation_id='inv-123', # Required
    author='agent_name',     # Required
    content=types.Content(...) # Required
)
```

### | Issue: "State not persisting across calls"

**Solution:** Use session:

```
# ✓ Use session for state persistence
session = Session()

result1 = runner.run(query1, agent=agent, session=session)
result2 = runner.run(query2, agent=agent, session=session) # State preserved

# ✗ No session - state lost
result1 = runner.run(query1, agent=agent)
result2 = runner.run(query2, agent=agent) # State reset
```

## Summary

You've mastered events and observability:

### Key Takeaways:

- ✓ `Event` class tracks all agent actions
- ✓ `EventActions` controls state, transfers, escalation
- ✓ `state_delta` for state modifications
- ✓ `artifact_delta` tracks file changes
- ✓ `escalate` for human review
- ✓ `transfer_to_agent` for agent handoffs
- ✓ ADK web trace view for visualization
- ✓ Custom logging and metrics for production monitoring

### Production Checklist:


- [ ] Events logged for all critical operations
- [ ] State changes tracked with `state_delta`
- [ ] Escalation rules defined and tested
- [ ] Monitoring dashboard configured
- [ ] Alerting rules for anomalies
- [ ] Trace view accessible for debugging
- [ ] Metrics collected (latency, errors, escalations)
- [ ] Event retention policy defined

**Next Steps:**

- **Tutorial 19:** Learn Artifacts & File Management
- **Tutorial 20:** Master YAML Configuration
- **Tutorial 21:** Explore Multimodal & Image Generation

**Resources:**

- [ADK Events Documentation](https://google.github.io/adk-docs/events/) (https://google.github.io/adk-docs/events/)
  - [Observability Guide](https://google.github.io/adk-docs/observability/) (https://google.github.io/adk-docs/observability/)
  - [ADK Web Interface](https://google.github.io/adk-docs/tools/adk-web/) (https://google.github.io/adk-docs/tools/adk-web/)
- 

 **Tutorial 18 Complete!** You now know how to implement comprehensive observability for production agents. Continue to Tutorial 19 to learn about artifact management.

---

Generated on 2025-10-21 09:02:41 from 18\_events\_observability.md

Source: Google ADK Training Hub