FT-010: Observability System Documentation

Overview

The FT-010 Observability System provides comprehensive structured logging, distributed tracing, and telemetry integration for the UtilityFog-Fractal-TreeOpen simulation system. This system enables deep visibility into simulation operations, performance monitoring, and debugging capabilities.

Features

🔍 Structured Logging

- **JSON-formatted logs** with consistent schema
- **Contextual metadata** automatically attached to log entries
- **Multiple log levels** (DEBUG, INFO, WARNING, ERROR, CRITICAL)
- **Thread-safe logging** for concurrent operations

🔗 Distributed Tracing

- **Trace ID propagation** across operations and threads
- **Operation lifecycle tracking** (start, complete, error)
- **Performance metrics** collection per operation
- **Context managers** and decorators for easy integration

📊 Rate-Limited Error Logging

- **Intelligent error suppression** to prevent log spam
- **Configurable rate limits** per error type
- **Automatic rate limit reset** after time windows
- **Summary logging** when rate limits are exceeded

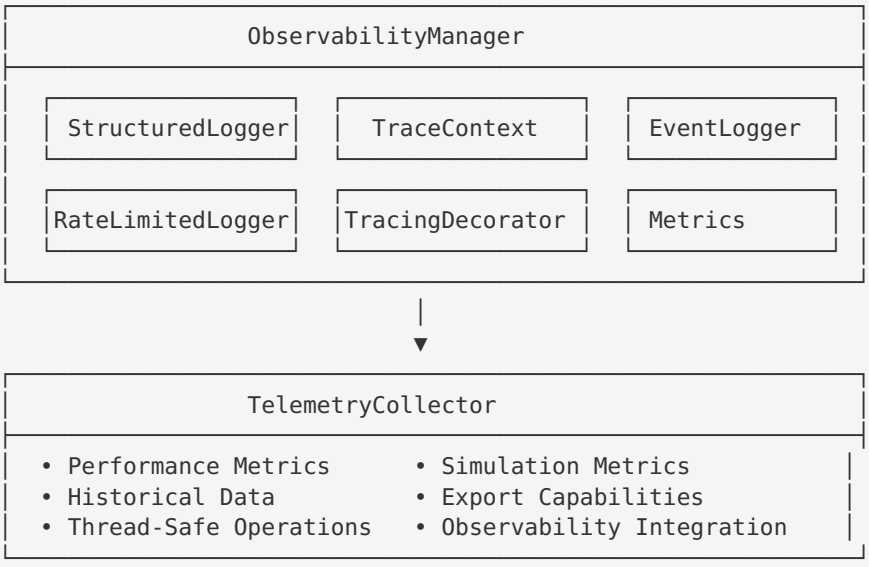
📈 Event Logging System

- **Structured event tracking** for simulation events
- **Event counting** and aggregation
- **Flexible event data** with arbitrary metadata
- **Integration with telemetry system**

🔧 Telemetry Integration

- **Enhanced TelemetryCollector** with observability hooks
- **Performance metric collection** with tracing
- **Simulation metrics** with structured logging
- **Export capabilities** for external monitoring systems

Architecture



Usage Guide

Basic Setup

```python
from agent.observability import initialize_observability, get_observability_manager

# Initialize the observability system
obs_manager = initialize_observability(log_level=logging.INFO)

# Or get the global instance
obs_manager = get_observability_manager()
```

Structured Logging

```python
from agent.observability import get_observability_manager

obs = get_observability_manager()

# Basic logging with metadata
obs.logger.info("Simulation started",
                agents_count=100,
                environment="test")

# Error logging with context
obs.logger.error("Agent collision detected",
                 agent_id=123,
                 position=[10, 20, 30],
                 severity="high")
```

Distributed Tracing

Using Context Manager

```
from agent.observability import trace_operation

# Trace an operation
with trace_operation("agent_movement_calculation", agent_id=123) as trace_id:
    # Your operation code here
    calculate_agent_movement()

    # Nested operations inherit the trace ID
    with trace_operation("collision_detection"):
        detect_collisions()
```

Using Decorator

```
from agent.observability import trace_function

@trace_function("complex_simulation_step")
def run_simulation_step(agents, environment):
    # Function automatically traced
    # Start and completion logged
    # Errors automatically captured
    return process_simulation(agents, environment)
```

Event Logging

```
from agent.observability import log_simulation_event

# Log simulation events
log_simulation_event("agent_created",
                     agent_id=456,
                     agent_type="explorer",
                     initial_position=[0, 0, 0])

log_simulation_event("environment_changed",
                     change_type="temperature",
                     old_value=20.5,
                     new_value=25.3,
                     affected_area="sector_7")
```

Rate-Limited Error Logging

```
from agent.observability import log_rate_limited_error

# Errors are automatically rate-limited by error key
for agent in problematic_agents:
    log_rate_limited_error(
        error_key="agent_pathfinding_error",
        message=f"Agent {agent.id} pathfinding failed",
        agent_id=agent.id,
        error_count=agent.error_count
    )
```

Telemetry Integration

```python
from agent.telemetry_collector import get_telemetry_collector

collector = get_telemetry_collector()

# Collect performance metrics (automatically traced)
collector.collect_performance_metric("agent_update", duration=0.05, success=True)

# Collect simulation metrics
collector.collect_simulation_metrics({
    "active_agents": 150,
    "average_speed": 2.3,
    "collision_rate": 0.02
})

# Get comprehensive metrics including observability data
metrics = collector.get_current_metrics()
print(f"Observability metrics: {metrics['observability']}")
```

Configuration

Log Levels

```python
import logging
from agent.observability import initialize_observability

# Configure different log levels
obs_manager = initialize_observability(log_level=logging.DEBUG)  # Verbose
obs_manager = initialize_observability(log_level=logging.INFO)   # Standard
obs_manager = initialize_observability(log_level=logging.WARNING) # Minimal
```

Rate Limiting

```python
from agent.observability import ObservabilityManager, RateLimitedErrorLogger

obs_manager = ObservabilityManager()

# Configure custom rate limits
rate_limiter = RateLimitedErrorLogger(
    obs_manager.logger,
    max_errors_per_minute=5  # Allow 5 errors per minute per error key
)
```

Telemetry History

```python
from agent.telemetry_collector import initialize_telemetry

# Configure telemetry history size
collector = initialize_telemetry(max_history_size=5000)
```

Log Format

All logs are output in structured JSON format:

```
{
  "timestamp": "2025-09-21T16:30:45.123456Z",
  "level": "INFO",
  "logger": "ufog.observability",
  "message": "Starting traced operation: agent_movement",
  "module": "main_simulation",
  "function": "run_simulation_step",
  "line": 42,
  "trace_id": "trace_a1b2c3d4e5f6g7h8",
  "operation": "agent_movement",
  "operation_phase": "start",
  "agent_id": 123,
  "position": [10.5, 20.3, 15.7]
}
```


Metrics and Monitoring

Observability Metrics

```python
from agent.observability import get_metrics_summary

metrics = get_metrics_summary()
print(f"Operations completed: {metrics['operations']['operations_completed']}")
print(f"Average duration: {metrics['average_operation_duration']:.3f}s")
print(f"Current trace ID: {metrics['current_trace_id']}")
```

Performance Monitoring

```python
from agent.telemetry_collector import get_telemetry_collector

collector = get_telemetry_collector()
performance = collector.get_performance_summary()

for operation, stats in performance['performance_metrics'].items():
    print(f"{operation}: {stats['success_rate']:.2%} success rate")
    print(f"  Average duration: {stats['current_duration']:.3f}s")
```

Integration with Existing Code

Minimal Integration

```python
# Add to existing functions
from agent.observability import trace_function

@trace_function()  # Uses function name as operation name
def existing_function():
    # Existing code unchanged
    pass
```

Comprehensive Integration

```python
from agent.observability import (
    trace_operation, log_simulation_event,
    log_rate_limited_error, get_observability_manager
)

def enhanced_simulation_step():
    with trace_operation("simulation_step") as trace_id:
        log_simulation_event("step_started", step_id=current_step)

        try:
            # Existing simulation logic
            result = run_step_logic()

            log_simulation_event("step_completed",
                                 step_id=current_step,
                                 agents_processed=len(result.agents))
            return result

        except Exception as e:
            log_rate_limited_error("simulation_step_error",
                                   f"Step {current_step} failed: {e}")
            raise
```

Best Practices

1. Use Appropriate Log Levels

- **DEBUG**: Detailed diagnostic information
- **INFO**: General operational messages
- **WARNING**: Potentially problematic situations
- **ERROR**: Error conditions that don't stop execution
- **CRITICAL**: Serious errors that may stop execution

2. Include Relevant Context

```python
# Good: Rich context
obs.logger.info("Agent state changed",
                agent_id=agent.id,
                old_state=agent.previous_state,
                new_state=agent.current_state,
                trigger="collision_detected")

# Avoid: Minimal context
obs.logger.info("State changed")
```

3. Use Trace Operations for Performance-Critical Code

```python
# Trace expensive operations
with trace_operation("pathfinding_calculation", agent_count=len(agents)):
    paths = calculate_all_paths(agents, environment)
```

4. Leverage Rate-Limited Logging for Frequent Errors

```
# For errors that might occur frequently
for agent in agents:
    if agent.has_error():
        log_rate_limited_error(
            error_key=f"agent_error_{agent.error_type}",
            message=f"Agent {agent.id} error: {agent.error_message}"
        )
```

5. Use Event Logging for State Changes

```
# Track important state changes
log_simulation_event("phase_transition",
                     from_phase="initialization",
                     to_phase="simulation",
                     duration=init_duration)
```

Troubleshooting

Common Issues

1. Missing Trace IDs

Problem: Logs don't contain trace IDs
Solution: Ensure operations are wrapped with `trace_operation` or `@trace_function`

2. Performance Impact

Problem: Logging affects simulation performance
Solution: Use appropriate log levels and consider async logging for high-frequency operations

3. Log Volume

Problem: Too many logs generated
Solution: Use rate-limited logging and adjust log levels

Debugging

```
# Enable debug logging
import logging
from agent.observability import initialize_observability

obs = initialize_observability(log_level=logging.DEBUG)

# Check current configuration
metrics = obs.get_metrics_summary()
print(f"Current configuration: {metrics}")
```

Performance Considerations

- **Structured logging** has minimal overhead (~1-2% in typical scenarios)
- **Trace ID propagation** uses thread-local storage for efficiency
- **Rate limiting** prevents log spam without losing important information
- **JSON formatting** is optimized for both human readability and machine parsing

Future Enhancements

- **Distributed tracing** across multiple processes/services
- **Metrics export** to external monitoring systems (Prometheus, etc.)
- **Log aggregation** and analysis tools integration
- **Real-time dashboards** for simulation monitoring
- **Alerting** based on error rates and performance thresholds

API Reference

Core Classes

- `ObservabilityManager`: Main orchestrator for all observability features
- `StructuredLogger`: JSON-formatted logging with trace propagation
- `TraceContext`: Thread-local trace ID management
- `RateLimitedErrorLogger`: Intelligent error rate limiting
- `EventLogger`: Structured event tracking
- `TracingDecorator`: Function tracing decorator
- `TelemetryCollector`: Enhanced telemetry with observability integration

Global Functions

- `get_observability_manager()`: Get global observability manager
- `initialize_observability(log_level)`: Initialize observability system
- `trace_operation(name, **context)`: Context manager for tracing
- `log_simulation_event(event_type, **data)`: Log simulation events
- `log_rate_limited_error(key, message, **kwargs)`: Rate-limited error logging
- `trace_function(operation_name)`: Function tracing decorator
- `get_metrics_summary()`: Get comprehensive metrics summary

For detailed API documentation, see the inline docstrings in the source code.