# Elizabeth Fuentes Leone

Developer Advocate, GenAI @ AWS

# Context Engineering

## Stop Agents from Choking on Their Own Data

### Solving Context Window Overflow in AI Agents

**Anton Bulle Labate**
IBM Research, Brazil
anton.labate1@ibm.com

**Valesca Moura de Sousa**
IBM Research, Brazil
valesca.sousa@ibm.com

**Sandro Rama Fiorini**
IBM Research, Brazil
srfiorini@ibm.com

**Leonardo Guerreiro Azevedo**
IBM Research, Brazil
lga@br.ibm.com

**Raphael Melo Thiago**
IBM Research, Brazil
raphaelt@br.ibm.com

**Viviane Torres da Silva**
IBM Research, Brazil
vivianet@br.ibm.com

# The Problem: Log Analysis System

An AI agent processes application logs to detect errors and anomalies:

**1 Fetch logs**

Tool returns 24 hours of events (~86,400 events, >5MB)

**2 Analyze patterns**

Requires complete dataset (indivisible, cannot truncate)

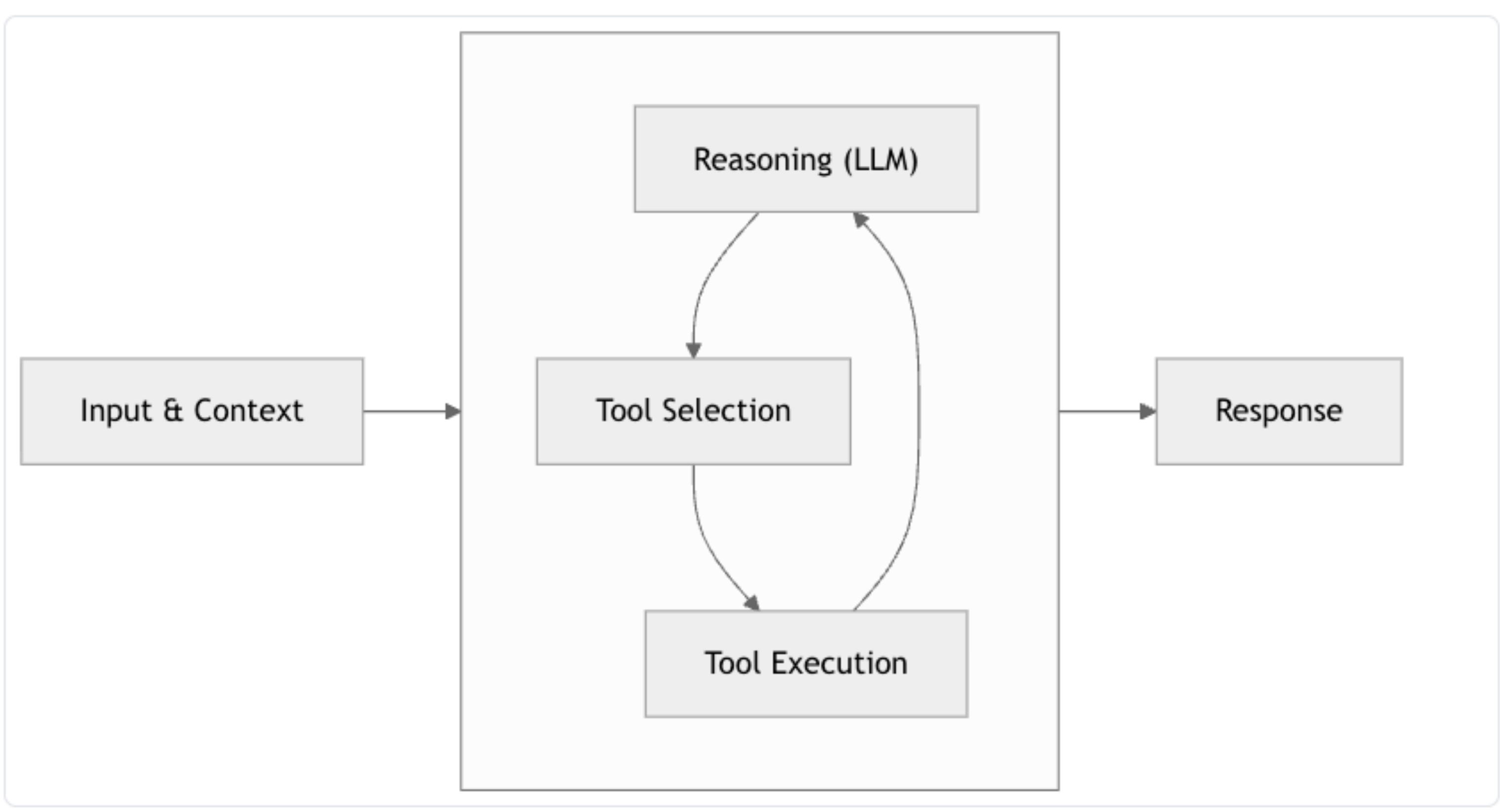
**3 Generate report**

Combines multiple analyses into incident summary

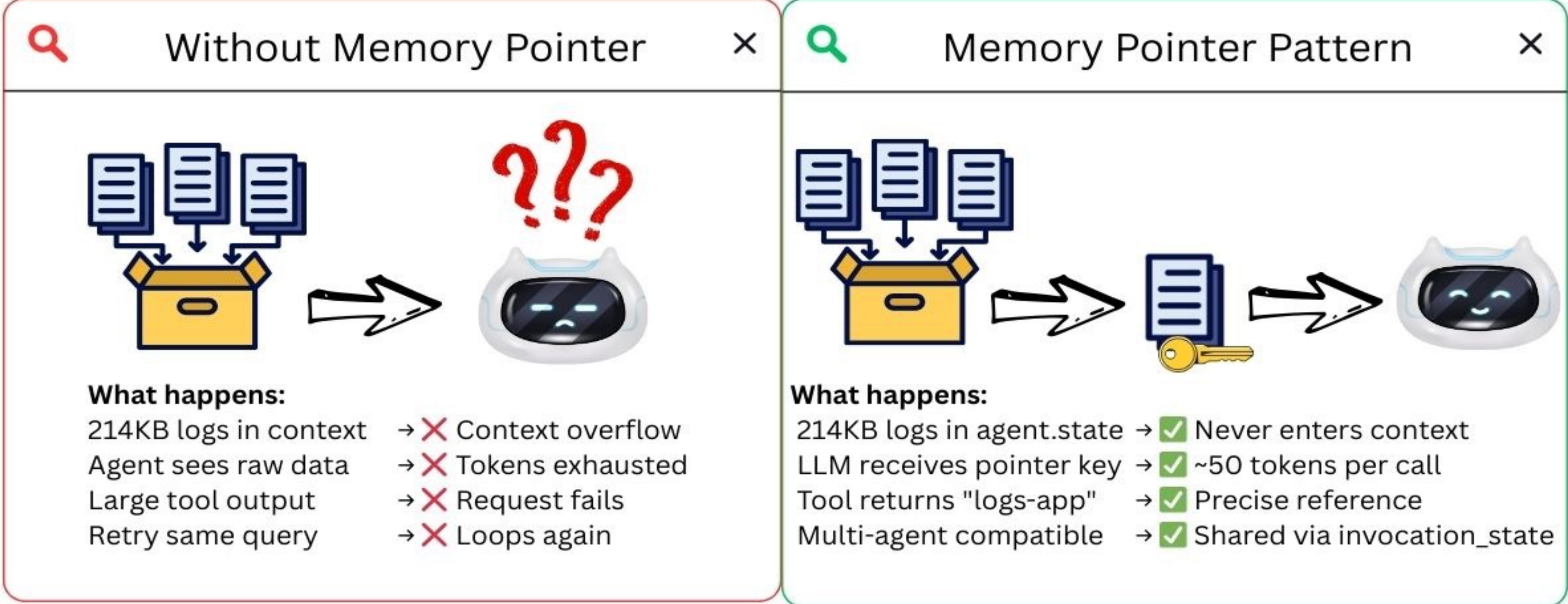
Logs cannot be truncated without losing critical events. Analysis requires the full dataset. A single tool call can return megabytes with no built-in size limit. Context overflow produces zero error messages.

# Agent Loop

When a model receives a request it cannot fully address with its training alone, it needs to reach out into the world: read files, query databases, call APIs, execute code.



## Without vs With Memory Pointer Pattern



| BEFORE | | AFTER |
| --- | --- | --- |
| **214KB** | → | **52 bytes** |
| Raw logs in context | | Pointer in context |
| 150,000+ tokens | | ~20,000 tokens |
| Overflow or degradation | | Full data preserved |

**7x token reduction (IBM Research)**

## Memory Pointer Pattern: Store + Return Pointer

```
@tool(context=True)
def fetch_application_logs(
    app_name: str,
    tool_context: ToolContext,
    hours: int = 24
) -> str:
    logs = generate_logs(hours)  # Large dataset

    if len(logs) > threshold:
        pointer = f"logs-{app_name}"
        tool_context.agent.state.set(pointer, logs)
        return f"Data stored at: {pointer}"

    return logs
```

# Memory Pointer Pattern: Resolve Pointer

```
@tool(context=True)
def analyze_error_patterns(
    logs_pointer: str,
    tool_context: ToolContext
) -> str:
    # Resolve pointer from agent.state
    logs = tool_context.agent.state.get(logs_pointer)

    # Analyze full dataset (never entered context)
    errors = [l for l in logs if l["level"] == "ERROR"]

    result = {
        "total_errors": len(errors),
        "unique_messages": len(set(e["message"] for e in errors))
    }
    return json.dumps(result)
```

## Swarm Multi-Agent: Collector → Analyzer → Reporter

**Collector**

fetch_application_logs()
Stores 145KB+ in
invocation_state

→

**Analyzer**

analyze_error_patterns()
Reads from shared state
Finds patterns

→

**Reporter**

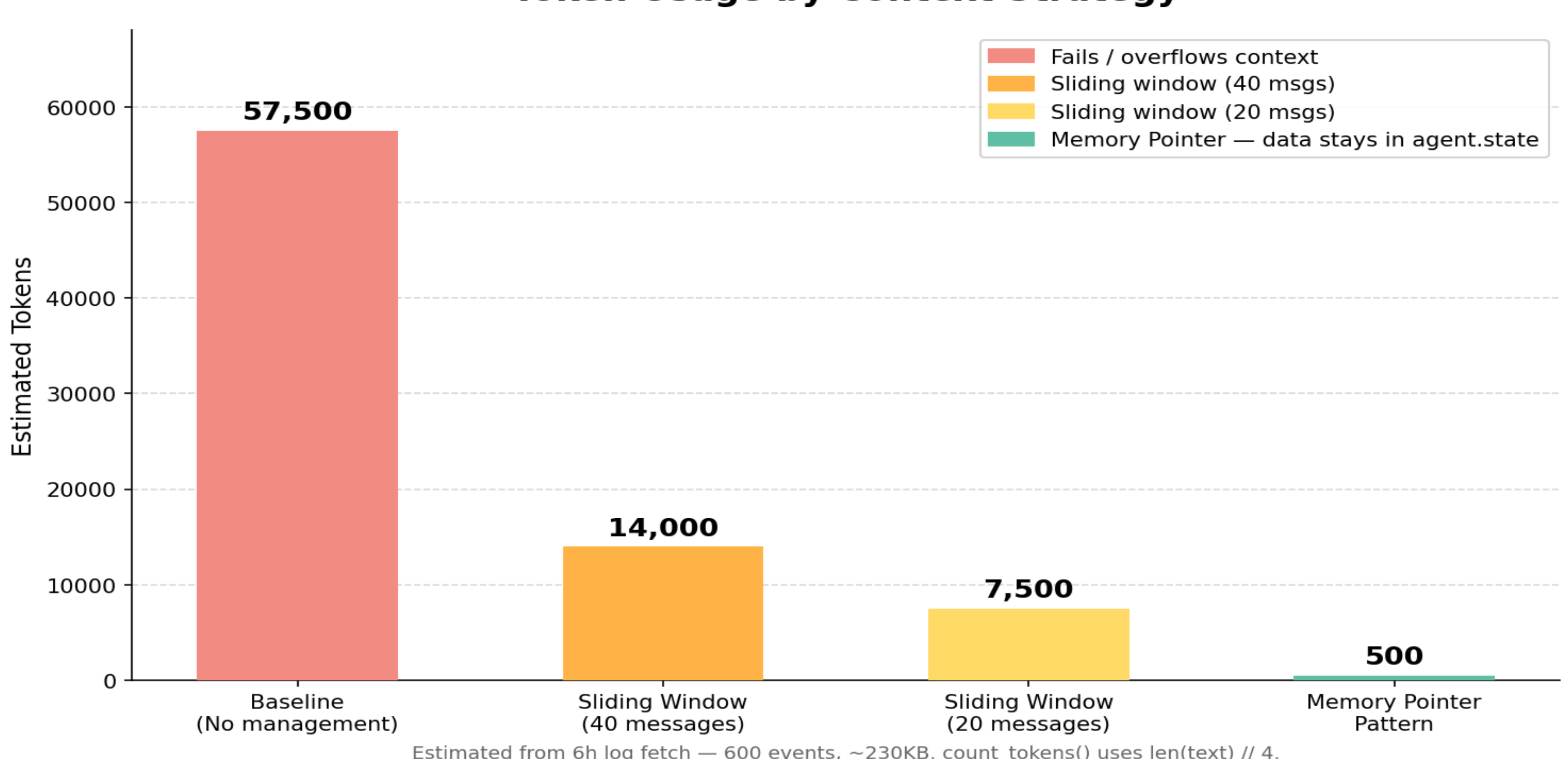
generate_incident_report(
)
Reads analysis results
Produces summary

tool_context.invocation_state (shared across all agents in the swarm)

Single-agent: agent.state

Multi-agent: invocation_state

# Token Usage by Context Strategy



# Five Scenarios Demonstrated

| # | Scenario | Description | Result |
|---|---|---|---|
| 1 | **Baseline** | No context management | **Fails or degrades** |
| 2 | **Memory Pointer** | IBM Research pattern | **7x token reduction** |
| 3 | **Custom Window** | Smaller window (20 msgs) | **Further optimization** |
| 4 | **Per-Turn** | Proactive management | **Complex workflows** |
| 5 | **Swarm Multi-Agent** | Collector → Analyzer → Reporter | **Autonomous coordination** |

## SlidingWindowConversationManager

```
from strands.agent.conversation_manager import (
    SlidingWindowConversationManager
)

agent = Agent(
    model=OpenAIModel(model_id="gpt-4o-mini"),
    conversation_manager=SlidingWindowConversationManager(
        window_size=40,  # Keep last 40 messages
        per_turn=True    # Apply every model call
    ),
    tools=[fetch_application_logs, analyze_error_patterns,
              generate_report]
)
```

Automatic trimming | Preserves tool pairs | Automatic retry on overflow | Per-turn management

# LIVE DEMO

1. python test_context_overflow.py (4 single-agent scenarios)
2. python swarm_demo.py (Collector → Analyzer → Reporter)
3. Compare: baseline fails vs pointer succeeds in ~14s

# Key Takeaways

1. Context overflow happens with large tool outputs and produces zero errors
2. Memory Pointer Pattern stores data in agent.state, returns lightweight pointers
3. Multi-agent systems share data via invocation_state, not context
4. SlidingWindowConversationManager provides automatic overflow protection
5. All patterns are transparent to the agent and framework-agnostic

# Resources

| | |
|---|---|
| **Code** | github.com/aws-samples/sample-why-agents-fail |
| **Paper** | Solving Context Window Overflow in AI Agents (IBM Research, arxiv 2511.22729) |
| **Blog** | dev.to/elizabethfuentes12 |
| **Docs** | strandsagents.com (Agent State, Conversation Management, Swarm) |

# Thank You!

Elizabeth Fuentes Leone

elifuentes.tech

Resources

bit.ly/this-event-resources

Blog & Socials

elifuentes.tech