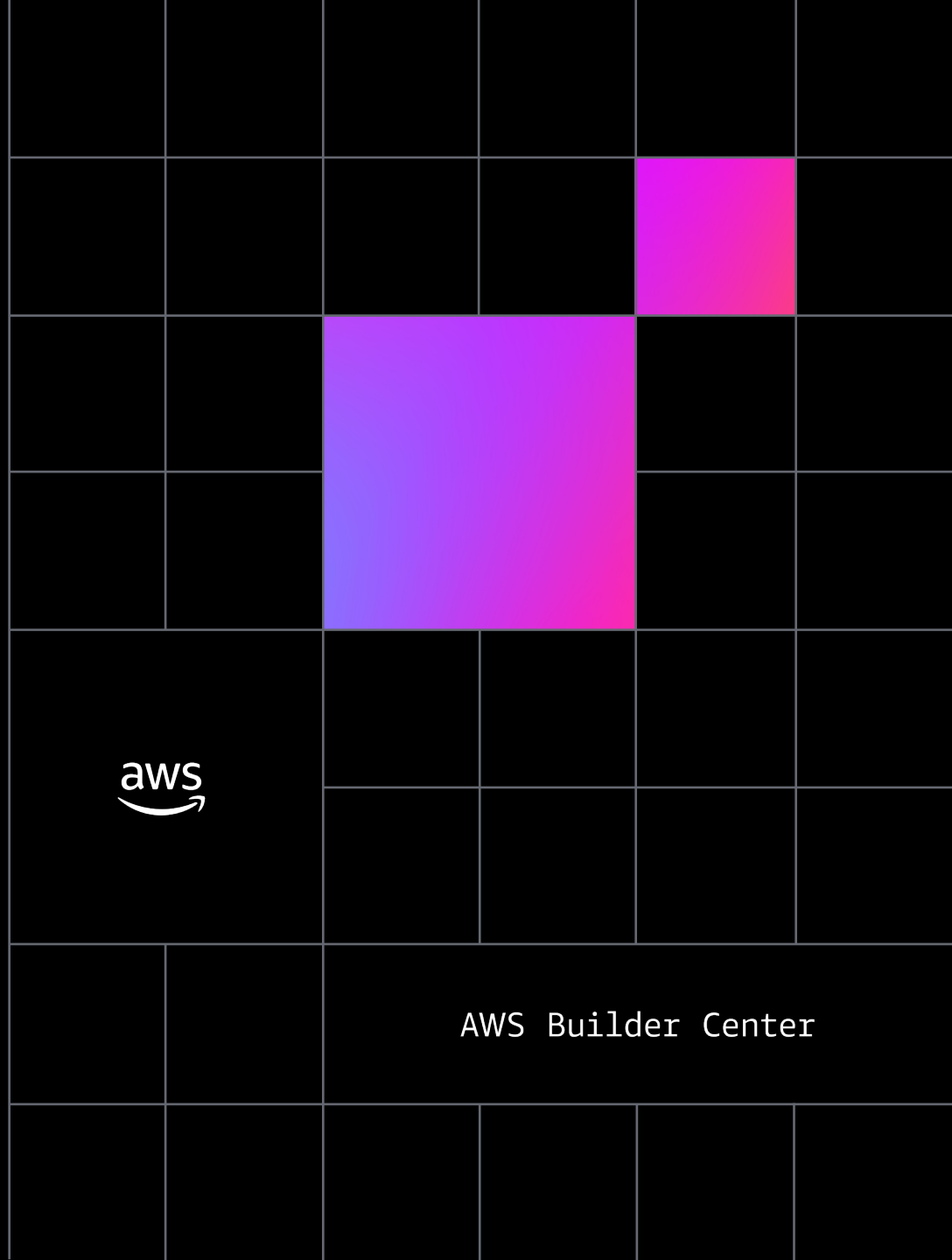
The Infinite Context Window Is a Myth:
Context Engineering for AI Agents

Elizabeth Fuentes Leone
Developer Advocate/SDE, GenAI

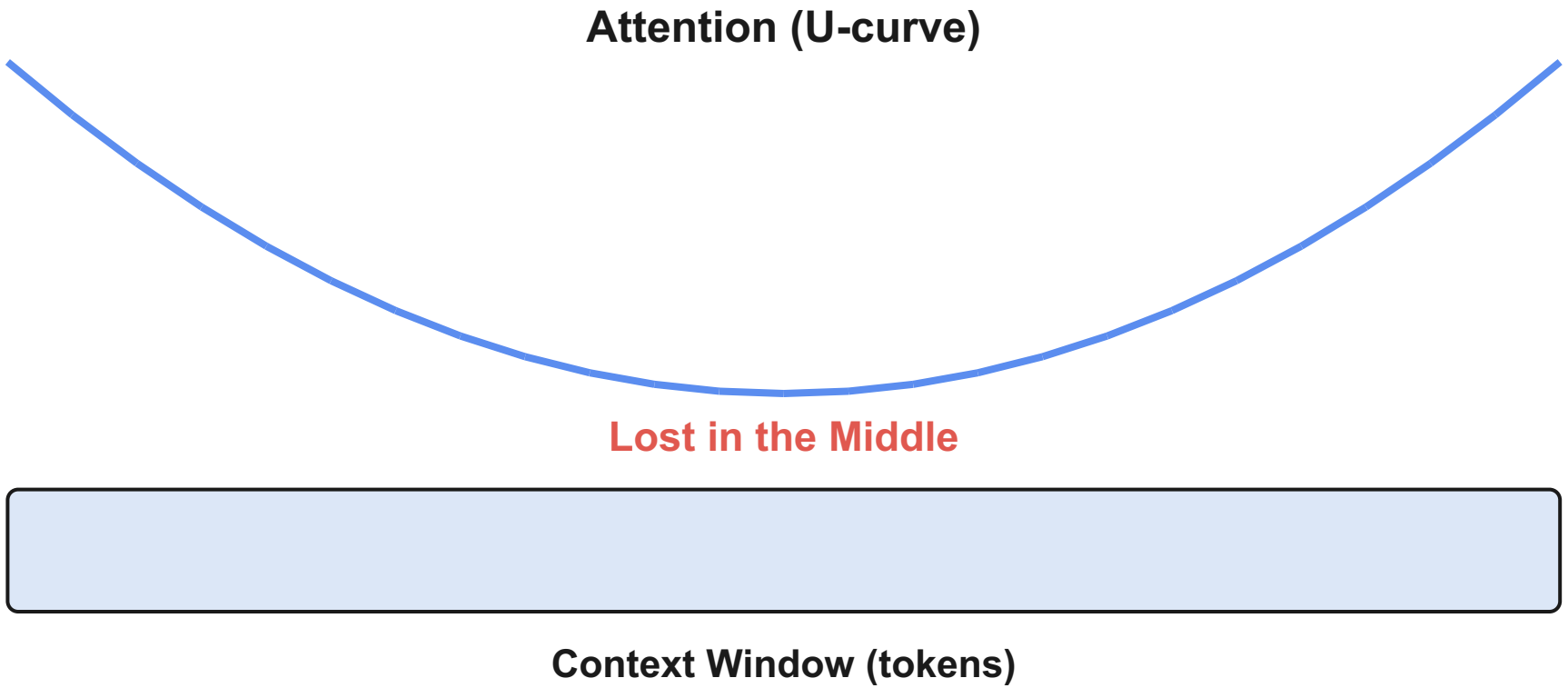
Morgan Willis
Sr. Developer Advocate, GenAI



4K → 32K → 128K → 1M → 2M tokens

“Just add more tokens, that'll fix it”

Why It Doesn't Work



More tokens ≠ more useful signal

Context Engineering

Giving the model the information it needs, when it needs it.

Curating the context window for optimal outcomes and token consumption.

Four Context Engineering Strategies

Externalize

Move large data to persistent storage

Select

Retrieve only what is relevant

Compress

Reduce tokens: summarize, compact

Isolate

Separate context across agents

The Starting Point: Conversation Management

Compress

Conversation management: the built-in default Strands Agents

The built-in default — before any advanced pattern

```python
from strands import Agent
from strands_tools import file_read, editor, shell

agent = Agent(
    system_prompt="You are a coding agent.",
    tools=(file_read, editor, shell)
)
```

STRANDS AGENTS SDK

Compress

```python
from strands import Agent
from strands.agent.conversation_manager import (
    SummarizingConversationManager)

agent = Agent(
    conversation_manager=SummarizingConversationManager(
        summary_ratio=0.5,          # summarize oldest 50%
        preserve_recent_messages=4,  # keep recent turns verbatim
    ),
)
```

STRANDS AGENTS SDK

Tiered Memory

Externalize

Compress

Short-term memory

- Recent conversation history
- Last k-turns always returned
- Sliding window + summarization

Long-term memory

- Extracted from conversation history
- Clean facts > raw conversation
- Scored retrieval

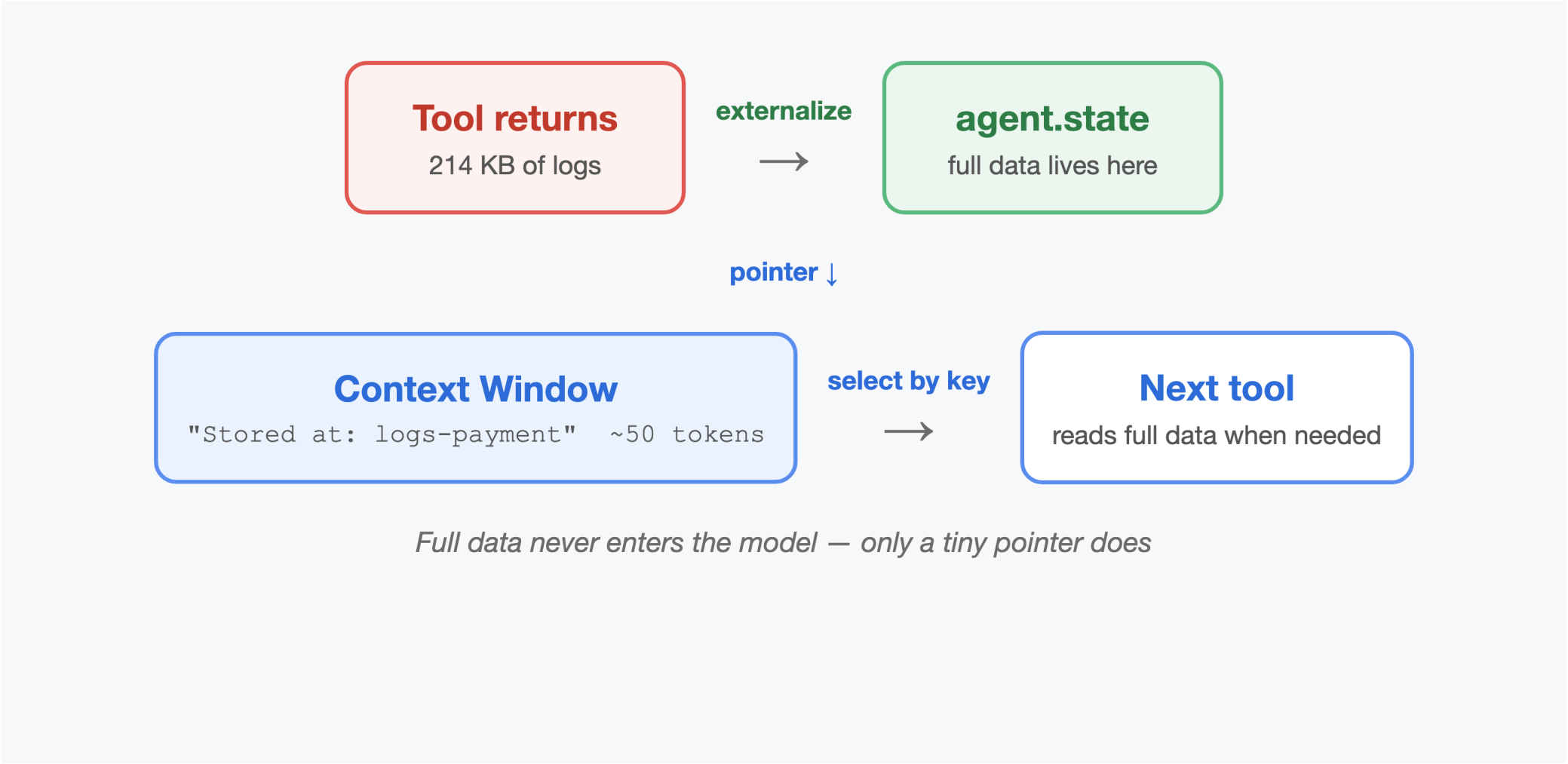
Relational memory

- Entity graph
- Traversal based
- How things connect

The Memory Pointer Pattern

Externalize Select

Single agent: log analysis via a pointer



Full data never enters the model — only a tiny pointer does


```python
@tool(context=True)
def fetch_application_logs(app_name, tool_context):
    logs = generate_logs(hours)       # ~214KB
    pointer = f"logs-{app_name}"
    # Externalize → agent.state
    tool_context.agent.state.set(pointer, logs)
    return f"Stored at: {pointer}"    # ~50 tokens

@tool(context=True)
def analyze_error_patterns(logs_pointer, tool_context):
    # Select → read full data only when needed
    logs = tool_context.agent.state.get(logs_pointer)
```

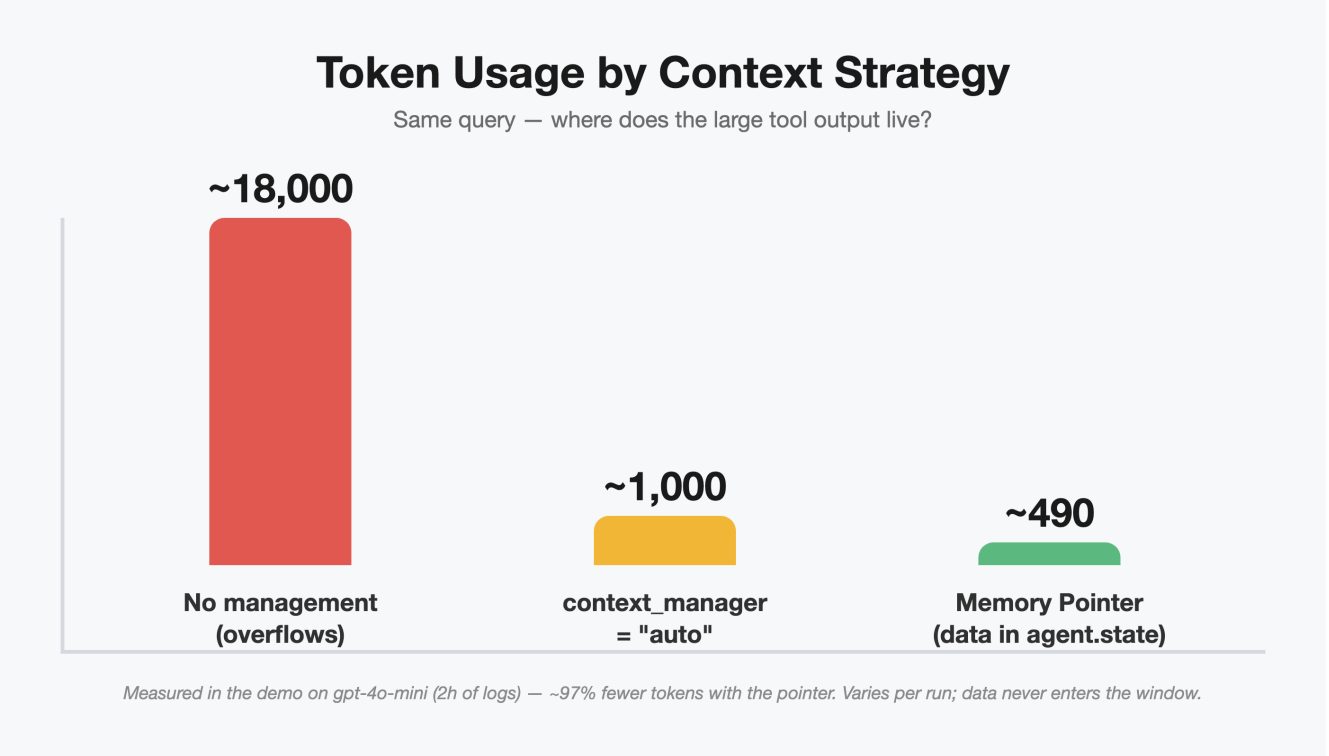
STRANDS AGENTS SDK

Result: Single Agent

~97%

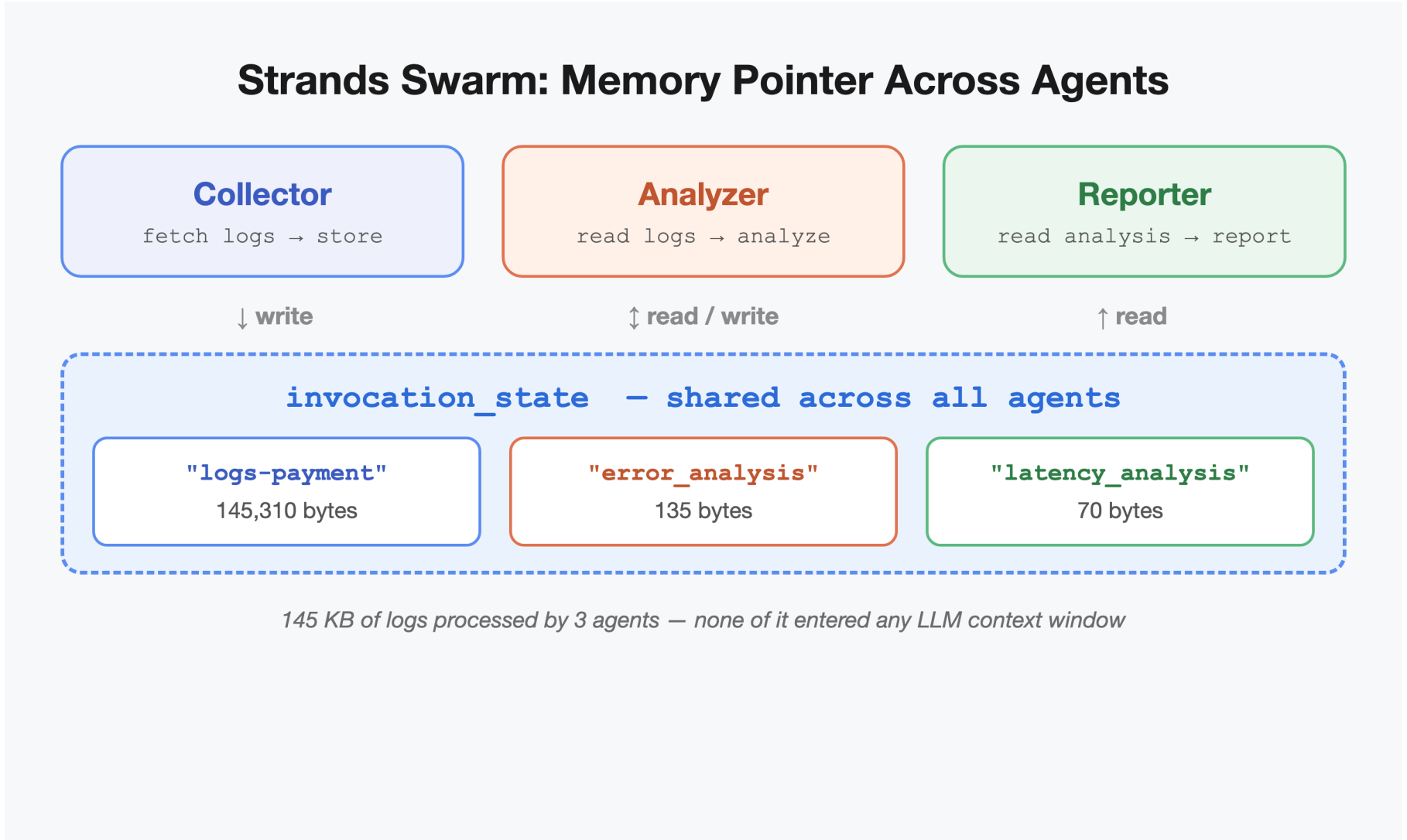
fewer tokens

with zero information loss



Now Scale It: One Pattern, Many Agents

Externalize Select Isolate




Externalize


Select


Isolate

Multi-agent swarm: shared pointer, scalable handoffs

```
@tool(context=True)
def fetch_logs_swarm(app_name, tool_context):
    logs = generate_logs(hours)              # 145KB+
    # Externalize into SHARED state
    tool_context.invocation_state[pointer] = logs
    return f"Stored as '{pointer}'. Hand off to analyzer."
collector = Agent(name="collector", tools=[fetch_logs_swarm])
analyzer  = Agent(name="analyzer",  tools=[analyze, latency])
reporter  = Agent(name="reporter",  tools=[generate_report])
swarm = Swarm([collector, analyzer, reporter])
```

STRANDS AGENTS SDK

Result: Multi-Agent Swarm


Externalize


Select


Isolate

3

agents coordinate
via one pointer

145 KB+

logs processed,
zero in any window

COMPLETED

collector → analyzer
→ reporter

Same pattern as one agent — now reliable, scalable orchestration

When the Agent Won't Stop

Reasoning loops, clear states + hard limit

Problem

- Tool says “more results may be available”
- Agent retries the same call
- Redundant calls, tokens burned, no progress

Clear States + Hard Limit

- Tools return SUCCESS / FAILED
- Agent stops on a clear signal
- LimitToolCounts caps the calls
- Hard ceiling regardless of the LLM

The Decoder: Language models can overthink

Compress

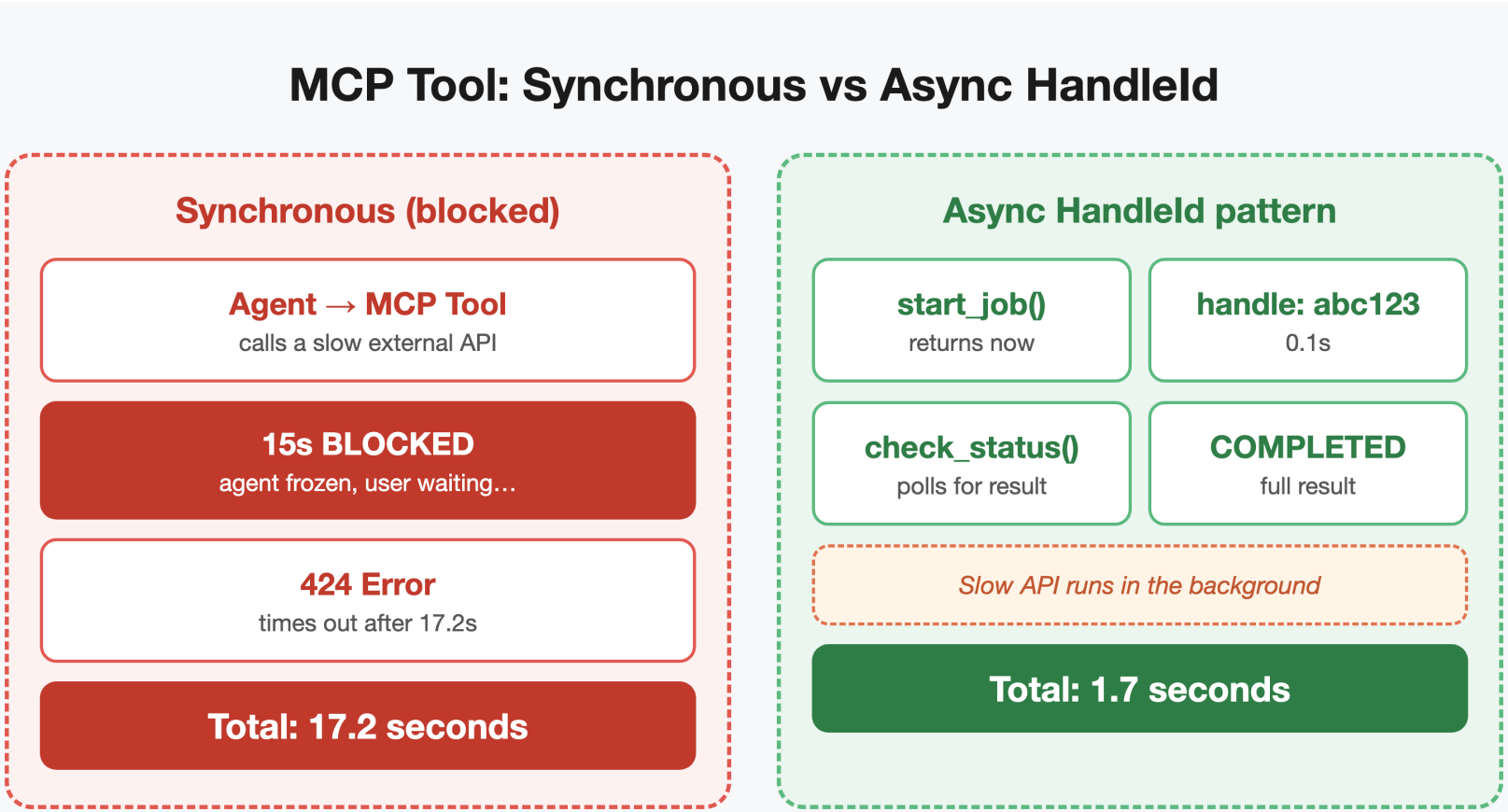
```python
@tool
def book_hotel(hotel, guest, nights):
    # Clear SUCCESS / FAILED ends the loop
    return f"SUCCESS: booking {conf} confirmed"


# LimitToolCounts — Strands Hooks Cookbook recipe
limit = LimitToolCounts(max_tool_counts={
    "search_flights": 3, "check_hotel_price": 3})
agent = Agent(tools=[...], hooks=[limit])
```

STRANDS AGENTS SDK

When a Tool Never Comes Back

MCP timeouts: async handle, no frozen agent

Isolate



Async handleId: the agent gets an immediate response and polls for results. No blocking, no 424 errors.

```python
@mcp.tool()
async def start_long_job(task: str) -> str:
    job_id = str(uuid.uuid4())[:8]
    # Isolate: run off the agent loop
    asyncio.create_task(process_job(job_id, task))
    return f"STARTED: {job_id}. Poll check_job_status."
@mcp.tool()
async def check_job_status(job_id: str) -> str:
    job = JOBS.get(job_id)
    return job["status"]  # running | completed: <result>
```

STRANDS AGENTS SDK

Result — MCP Timeouts

Isolate

17.2s → 1.7s

response time · no more 424 timeouts

Anti-Patterns

Context Stuffing

- Include everything and the kitchen sink
- Always reach for the largest window

Naive Summarization

- Important details get dropped
- Performance quietly goes down

Context Pollution

- Pass everything between agents
- One agent does everything

Match the Strategy to the Failure

Tool returns large data (logs, docs, datasets)

Externalize + Select — memory pointer

Many agents need the same large data

Shared invocation_state + scoped context

Agent loops, or a tool hangs

Clear states / LimitToolCounts / async handle

Thank you!

Elizabeth Fuentes · Morgan Willis

Developer Advocate, GenAI · Sr. Developer Advocate, GenAI

Try it yourself — all demos in Strands Agents

Resources

bit.ly/4xaaFMv

Blog & Socials

elifuentes.tech