AgentCore Memory: Short-Term Memory

Overview

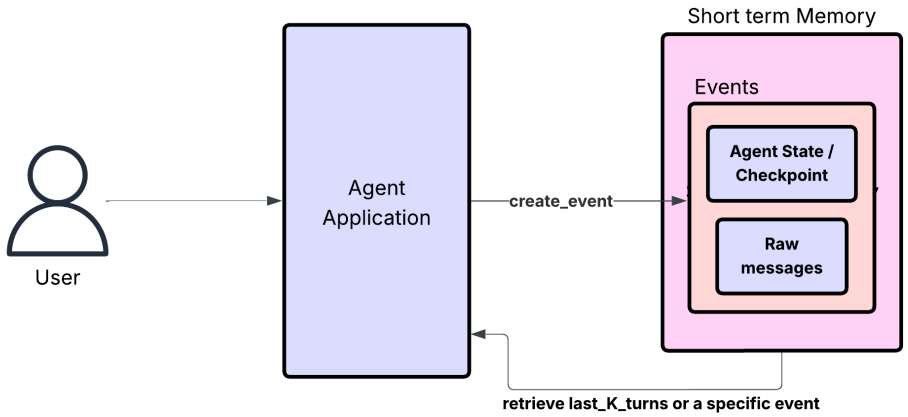
Short-term memory in Amazon Bedrock AgentCore provides immediate conversation context and session-based information management. It enables AI agents to maintain continuity within a single interaction or closely related sessions (with the same SessionID), ensuring coherent and contextually aware responses throughout a conversation.

What is Short-Term Memory?

Short-term memory focuses on:

- **Session Continuity**: Maintaining context within a single conversation or a session
- **Immediate Context**: Preserving recent conversation history for coherent responses
- **Intermediate State/Checkpointing**: Managing relevant transient information and metadata for the current interaction.

How Short-Term Memory Works in AgentCore



Event Storage

Events are the fundamental units of short-term memory stored as key-value pairs in AgentCore Memory. When we create an event using `create_event()`, it's scoped under:

- **memoryId**: Defined memory resource.
- **actorId**: Identifies the entity associated with the event, such as userId or agentId or agent/user combinations
- **sessionId**: Groups related events together, such as a conversation session

An event can take in user-agent message pairs or Agent State / Checkpoint. Events can be configured for automatic cleanup of expired session data (via the TTL parameter).

Session Management

Short-term memory operates at the session level. AgentCore Memory allows:

1. Retrieval of last `k` User/Agent message-pairs using `get_last_k_turns` or `list_events` SDK features.
2. Storing conversation metadata (timestamps, session IDs, actor IDs)
3. Maintaining its own context by saving Agent State, event timestamps, and User-Agent Conversation data

Real-Time Access

Short-term memory provides immediate access to stored events via `list_events` SDK feature. It's important to keep conversation continuation when a session discontinues or the agent fails and ensure real-time context updates as conversations progress

Implementation

```
client = MemoryClient(region_name="us-east-1")

# Create memory resource
memory = client.create_memory_and_wait(
    name="MyMemory", # Assigned Memory Name
    strategies=[], # Only needed for Long Term memory
    description="Memory for my agent",
    event_expiry_days=90, # Event Expiry time limit
)
memory_id = memory.get("id") # This ID is used in all subsequent memory operations

memory.create_event(
    memory_id=memory_id,
    actor_id="user_ABC",
    session_id="session_i", # Unique session Identifier
    messages=[
        ("Hi, I'm having trouble with my order #12345", "USER"),
        ("I'm sorry to hear that. Let me look up your order.", "ASSISTANT")]
    ) # Store one or more conversation turn
```

Integration with Frameworks

Short-term memory integrates seamlessly with popular agentic frameworks:

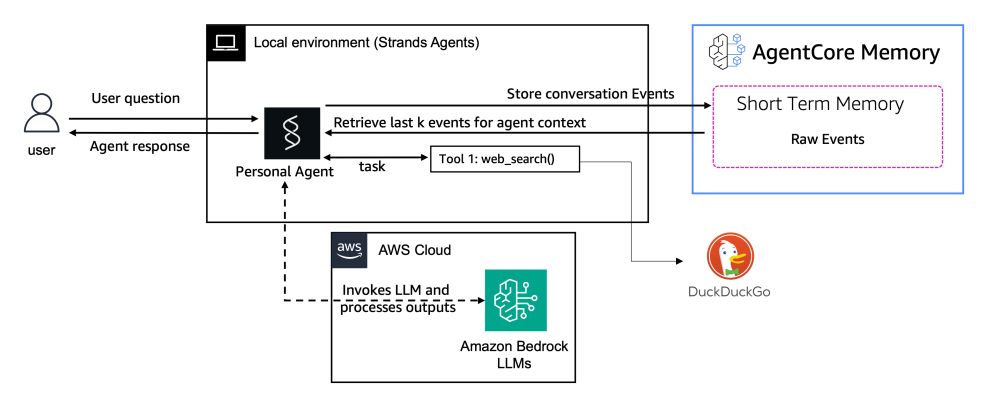
- **Strands Agent**: Native integration with conversation hooks
- **LangGraph**: State management integration
- **Custom Frameworks**: Direct API access for flexible implementation

Implementation Approaches

In this workshop, we'll explore two different implementation approaches for short-term memory: a single-agent approach using hooks and using memory tool provided to the agent.

Short Term Memory with Hooks (Recommended Approach)

Hooks allow us to add custom logic or functionality to an agent without directly modifying its core code. Using Hooks for memory keep the memory storage and retrieval more deterministic. The agent automatically retrieves recent conversations(last_k_turns) for context loading during agent Initialization. It also stores every user-agent message turn as soon as the agent generates a response for the user. Let's look at one example:



Here the **"Store conversation events"** and **"Retrieve last_k events"** are implemented using hooks via the HookProvider class from Strands. Refer to step 4 of this notebook.

Short Term Memory with Memory tool

Another approach to implementing short-term memory uses the Tool Use instead of custom hooks. In this case, we provide a memory tool implementation here to the agent and let the agent make the decision to store or retrieve the memories.

[Additional] Short Term Memory with Multi-Agent Architecture

While single-agent approaches focus on individual conversation context, multi-agent systems enable specialized agents to work together while potentially accessing a common memory store. The example usecase implemenation showcases how to implement a **multi-agent system with same memory** using AWS AgentCore Memory and the Strands framework. This approach demonstrates:

- **Agent Specialization**: Creates dedicated agents for specific domains or taks.
- **Namespace Isolation**: Each agent has its own unique namespace within the same memory
- **Tool-Based Architecture**: Specialized agents are implemented as tools for the coordinator
- **Memory Persistence Across Agents**: Information stored by one agent can be retrieved by the coordinator agent.

Try it out

At an AWS Event

If you are following the workshop via workshop studio, now go to JupyterLab in SageMaker Studio. In the JupyterLab UI navigate to the folder `04-AgentCore-memory/01-short-term-memory/` and based on your interest try out the following:

- Single Agent with Strands Agent (Personal Agent usecase): `01-single-agent/with-strands-agent/personal-agent.ipynb`
- Multi Agent with Strands Agent (Travel Planning): `02-multi-agent/with-strands-agent/travel-planning-agent.ipynb`

Self-paced

Here are notebooks that let you try out the implementations:

Framework	Use Case	Description	Notebook	Architecture
Strands Agent	Personal Agent	AI assistant that maintains conversation context and remembers user interactions within a session	personal-agent.ipynb	View

Framework	Use Case	Description	Notebook	Architecture
Strands Agent	Travel Planning	Multiple agents that share the same memory resource.	travel-planning-agent.ipynb	View

Congratulations ! Now you can use AgentCore Short-term Memory for your usecase !!

Previous Next