AgentCore Memory: Long-Term Memory Strategies

Introduction

Long-term memory in Amazon Bedrock AgentCore enables AI agents to maintain persistent information across multiple conversations and sessions. Unlike short-term memory that focuses on immediate context, long-term memory extracts, consolidates and stores meaningful information that can be retrieved and applied in future interactions, creating truly personalized and intelligent agent experiences.

AgentCore Long-term memory provides:

- **Cross-Session Persistence**: Information that is important beyond individual conversations
- **Intelligent Extraction**: Automatic identification and storage of important facts, preferences, and patterns
- **Semantic Understanding**: Vector-based storage that enables natural language retrieval
- **Personalization**: User-specific information that enables tailored experiences
- **Knowledge Accumulation**: Enabling Agent to learn and build information over time.

Why is this important?

Long-term memory transforms simple chatbots into intelligent assistants that truly understand their users. This capability is critical for:

1. **Creating Personalized Experiences** - Agents remembers user preferences, conversation history, and specific details, allowing them to provide tailored responses without requiring users to repeat information, thus creating a natural conversational flow.
2. **Building Trust Through Continuity** - By maintaining consistent knowledge across interactions, agents appear more reliable and trustworthy to users.
3. **Session Summarization** - Long-term memory allows agents to have awareness of multiple past sessions.
4. **Continuous Improvement** - Enables Agent to learn and build information over time about specific users and domains.

Without long-term memory, agents are essentially starting fresh with each interaction, unable to build upon past conversations or truly understand the user needs over time.

What you will learn/build

In this section, you'll learn how to implement one or more long-term memory strategies in your AgentCore applications, creating intelligent agents that remember user preferences, extract important information, and maintain context across multiple sessions.

High-Level Architecture

Long-term memory operates through **Memory Strategies** that define what information to extract and how to process it. The system works automatically in the background to:

1. **Extract Information**: Based on the configuration such as Important data (facts, preferences, summaries) is extracted using AI models.
2. **Hierarchical Storage**: Extracted information is organized in namespaces for efficient retrieval

3. **Embedding & Indexing**: Information is vectorized for natural language search capabilities
4. **Consolidation**: Similar information is merged and refined over time

Processing Time: Typically takes ~30-40 seconds after conversations are saved, with no additional code required.

Understanding Namespaces

Namespaces organize memory records within strategies using a path-like structure. They can include variables that are dynamically replaced:

- `support/facts/{sessionId}`: Organizes facts by session
- `user/{actorId}/preferences`: Stores user preferences by actor ID
- `meetings/{memoryId}/summaries/{sessionId}`: Groups summaries by memory

The `{actorId}`, `{sessionId}`, and `{memoryId}` variables are automatically replaced with actual values when storing and retrieving memories.

AgentCore Memory Feature - Relevant Code

AgentCore Memory supports four distinct strategy types for long-term information storage. Here's a code example of how to create a memory instance with two strategies:

```python
from bedrock_agentcore.memory import MemoryClient
from bedrock_agentcore.memory.constants import StrategyType

# Initialize Memory Client
client = MemoryClient(region_name="us-east-1")

# Define memory strategies
strategies = [
    {
        StrategyType.SEMANTIC.value: {
            "name": "FactExtractor",
            "description": "Extracts and stores factual information",
            "namespaces": ["support/user/{actorId}/facts"]
        }
    },
    {
        StrategyType.USER_PREFERENCE.value: {
            "name": "UserPreferences",
            "description": "Captures user preferences",
            "namespaces": ["support/user/{actorId}/preferences"]
        }
    }
]

# Create memory resource
memory = client.create_memory_and_wait(
    name="CustomerSupportMemory",
    strategies=strategies,
    description="Memory for customer support agent",
    event_expiry_days=90,
)
memory_id = memory['id']  # This ID is used in all subsequent memory operations
```

The memory_id is a critical identifier used throughout the application:

- With **hooks** for deterministic context injection and conversation saving [recommended]
- With **memory tools** for agentic memory retrieval and storage
- For managing memory lifecycle (retrieving strategies, updating configuration, deletion)

Hooks and tools use this ID to identify which memory resource to interact with, enabling the agent to maintain context across multiple conversations.

Let's explore each strategy type in detail:

1. Semantic Memory Strategy

Stores factual information extracted from conversations using vector embeddings for similarity search.

```
{
    "semanticMemoryStrategy": {
        "name": "FactExtractor",
        "description": "Extracts and stores factual information",
        "namespaces": ["support/user/{actorId}/facts"]
    }
}
```

Best for: Storing any factual data that needs to be retrieved through natural language queries.

2. Session Summary Memory Strategy

Creates and maintains summaries of conversations to preserve context for long interactions. The sessionId parameter is mandatory for this strategy.

```
{
    "summaryMemoryStrategy": {
        "name": "ConversationSummary",
        "description": "Maintains conversation summaries",
        "namespaces": ["support/summaries/{sessionId}"]
    }
}
```

Best for: Providing context in follow-up conversations and maintaining continuity across long interactions.

3. User Preference Memory Strategy

Tracks user-specific preferences and settings to personalize interactions.

```
{
    "userPreferenceMemoryStrategy": {
        "name": "UserPreferences",
        "description": "Captures user preferences and settings",
        "namespaces": ["support/user/{actorId}/preferences"]
    }
}
```

Best for: Storing communication preferences, product preferences, or any user-specific settings.

4. Custom Memory Strategy

Allows customization of prompts for extraction and consolidation, providing flexibility for specialized use cases.

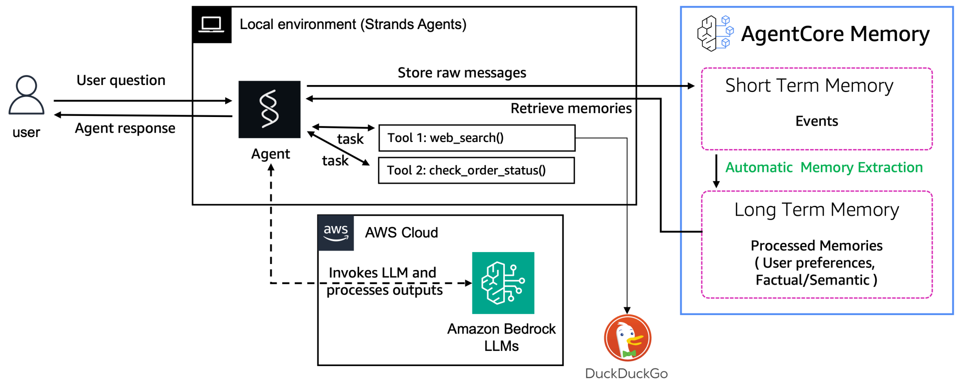
```
{
    "customMemoryStrategy": {
        "name": "CustomExtractor",
        "description": "Custom memory extraction logic",
        "namespaces": ["user/custom/{actorId}"],
        "configuration": {
            "semanticOverride": { # You can also override Summary or User Preferences.
                "extraction": {
                    "appendToPrompt": "Extract specific information based on custom criteria",
                    "modelId": "anthropic.claude-3-sonnet-20240229-v1:0",
                },
                "consolidation": {
                    "appendToPrompt": "Consolidate extracted information in a specific format",
                    "modelId": "anthropic.claude-3-sonnet-20240229-v1:0",
                }
            }
        }
    }
}
```

Best for: Specialized extraction needs that don't fit the standard strategies.

Different Implementation Approaches

AgentCore Memory offers multiple implementation approaches to suit different application needs. The following are examples that can be explored in this workshop:

Customer Support with Hooks

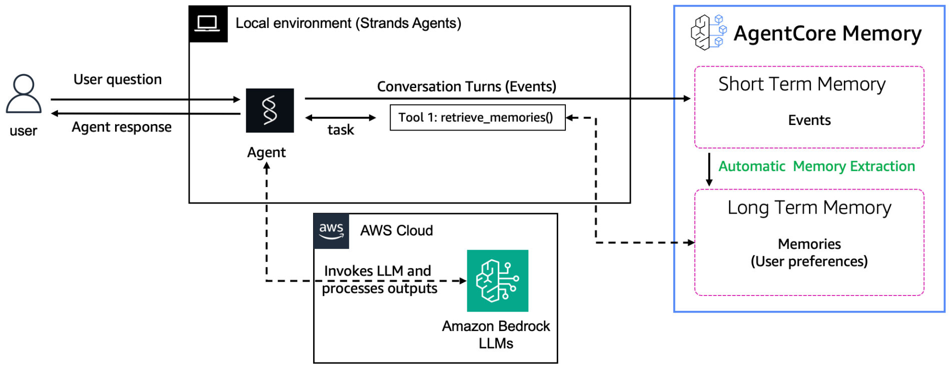


The Customer Support implementation demonstrates how to build an intelligent customer support agent with long-term memory using Strands Agent Hooks. This approach showcases:

- **Multiple Memory Strategies**: Uses both USER_PREFERENCE and SEMANTIC memory strategies to capture a comprehensive customer profile
- **Deterministic Context Injection**: Retrieves relevant customer context before processing new queries
- **Specialized Namespace Structure**: Organizes memories into separate namespaces for preferences and semantic facts
- **Custom Hook Implementation**: Defines hooks for both retrieving customer context and saving support interactions

This notebook shows how long-term memory can significantly enhance customer support scenarios by maintaining persistent records of customer preferences, order history, and past interactions.

Culinary Assistant with Tool-based Memory

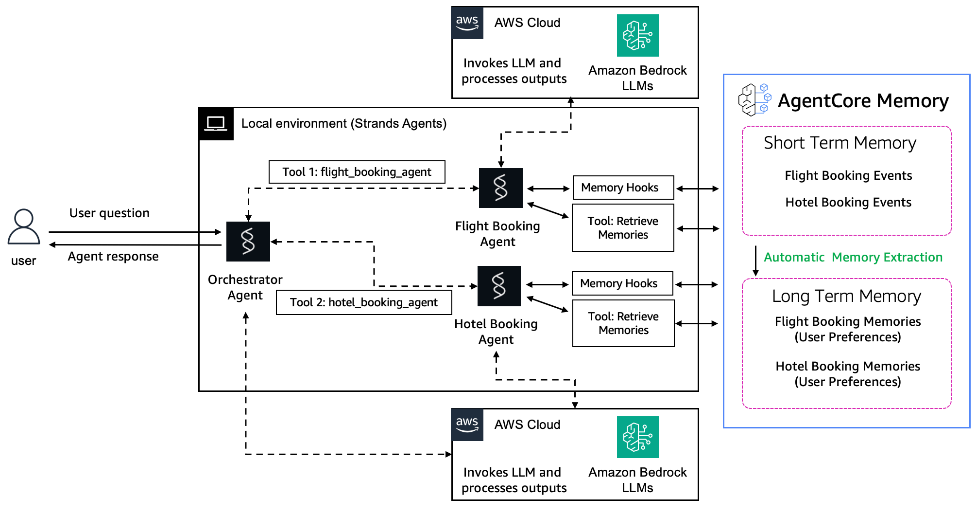

Local environment (Strands Agents)
User question
Agent response
user
Agent
Conversation Turns (Events)
task
Tool 1: retrieve_memories()
AWS Cloud
Invokes LLM and processes outputs
Amazon Bedrock LLMs
AgentCore Memory
Short Term Memory
Events
Automatic Memory Extraction
Long Term Memory
Memories (User preferences)

The Culinary Assistant implementation uses the AgentCoreMemoryToolProvider instead of custom hooks, demonstrating a simpler tool-based approach:

- **Single Memory Strategy**: Uses only USER_PREFERENCE strategy for simplicity
- **Tool-Based Implementation**: Uses AgentCoreMemoryToolProvider for built-in memory tools
- **Explicit Memory Access**: Agent explicitly calls memory tools when needed, which will not be deterministic
- **Hydrated Short-Term Memory**: Shows how to seed memory with previous conversation history
- **Beginner-Friendly Design**: More straightforward implementation for those new to AgentCore Memory

This approach is ideal for simpler use cases where the deterministic context injection of hooks might be unnecessary, or when you want the agent to have explicit control over when and how memory is accessed.

Travel Booking with Multi-Agent Architecture



The Travel Booking Assistant implementation showcases a more advanced architecture using multiple specialized agents that share a common memory infrastructure. This approach demonstrates:

- **Agent Specialization**: Creates dedicated agents for flight booking and hotel booking
- **Namespace Isolation**: Each agent has its own unique namespace within the shared memory
- **Coordinator Pattern**: Implements a coordinator agent that delegates queries to specialized agents
- **Hybrid Memory Implementation**: Combines memory tools for agentic access with hooks for deterministic context management
- **Memory Persistence Across Agents**: Information stored by one agent can be retrieved by others

This implementation shows how memory persists across different agent instances and how to create a cohesive user experience with specialized knowledge domains, leveraging both tools and hooks for optimal memory management.

Implementation Comparison

Notebook	Memory Strategy	Implementation Method	Context Retrieval	Key Featur
Customer Support	Multiple (USER_PREFERENCE and SEMANTIC)	Custom hook provider	Deterministic retrieval before processing	Deterministic context injection
Culinary Assistant	Single (USER_PREFERENCE)	AgentCoreMemoryToolProvider	Explicit tool calls	Simple tool-based implementat
Travel Booking	USER_PREFERENCE with multiple namespaces	AgentCoreMemoryToolProvider and hooks	Tool-based retrieval when needed	Multi-agent, shared memo

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 `04-AgentCore-memory/02-long-term-memory`:

- Strands Agents with Hooks (Customer Support): `01-single-agent/using-strands-agent-hooks/customer-support/customer-support.ipynb`
- Strands Agents with Tool (Culinary Assistant): `01-single-agent/using-strands-agent-memory-tool/culinary-assistant.ipynb`
- Multi-agent Strands Agents with hooks (Travel Booking): `02-multi-agent/with-strands-agent/travel-booking-assistant.ipynb`

Self-paced

Here are notebooks that let you try out the implementations:

- Customer Support with Long-Term Memory
- Math Assistant with Custom Memory Strategy
- Culinary Assistant with Tool-based Memory
- Multi-Agent Travel Booking with Shared Memory

Congratulations!

Congratulations! You've learned how to implement long-term memory strategies in Amazon Bedrock AgentCore. With these powerful capabilities, you can now create truly intelligent agents that remember user preferences, maintain context across sessions, and provide personalized experiences. A complete implementation transforms a simple chatbot into a personal assistant that grows more helpful with each interaction, remembering important details about users and adapting to their needs

Previous Next