Architecting Persistent and Adaptive AI Agents: A Guide to Advanced Memory Management

The development of sophisticated artificial intelligence (AI) agents necessitates robust memory management systems to enable coherent, personalized, and efficient interactions. Without the capacity to retain and recall information from past exchanges, AI agents operate in a perpetual state of amnesia, treating each user query as an isolated event. This fundamental limitation transforms potentially intelligent systems into what are often described as "expensive parrots," capable of generating plausible responses but lacking the contextual understanding to be truly useful. This report details the critical aspects of memory management for AI agents, updating foundational concepts and practical implementations to reflect the latest advancements in leading AI frameworks as of mid-2025.

Executive Summary

This report offers a comprehensive guide to implementing robust memory management in AI agents, updating the concepts and code from a seminal 2025 article to reflect the latest advancements in AI frameworks and underlying technologies. The core challenge addressed is the inability of AI agents to maintain context and continuity across interactions, leading to frustrating and inefficient user experiences. The analysis explores and modernizes implementations for both short-term (conversational context) and long-term (cross-session learning) memory using leading frameworks: LangChain, Pydantic AI, and Agno. The report highlights the evolution towards modern architectural patterns like LangChain Expression Language (LCEL) and LangGraph for enhanced modularity and performance. Beyond foundational memory, the discussion delves into advanced patterns such as multi-modal memory, intelligent compression, collaborative "Team Memory," and critical production considerations including scalable storage, advanced monitoring, cost optimization, and robust security practices. All presented code examples are updated to their latest stable versions, utilizing models like gpt-4o for OpenAI and asynchronous clients for persistent storage, ensuring practical applicability for developers building real-world AI agent systems.

1. Introduction: The Imperative of Memory in AI Agents

The effectiveness of an AI agent is profoundly tied to its ability to remember. Without a functional memory system, an AI agent cannot maintain context, learn from past interactions, or build a relationship with its users. This deficiency leads to repetitive questions, irrelevant responses, and ultimately, a frustrating user experience that undermines the utility of the AI.

1.1. The "Expensive Parrot" Problem

The fundamental challenge in building AI agents that deliver meaningful value stems from their inherent statelessness. When an agent lacks memory, every interaction begins anew, devoid of any prior context. This behavior is akin to conversing with an individual suffering from amnesia or, as the original article aptly describes, an "expensive parrot". Such an agent might generate grammatically correct and superficially plausible responses, but its inability to recall previous turns in a conversation renders it largely ineffective and inefficient.
Consider the following illustrative code, which embodies this stateless problem:

Python

```python
import openai # Assuming openai library is installed
# Ensure OPENAI_API_KEY is set in environment variables.
# For this 'broken' example, we simulate the core issue of statelessness.

# Initialize the OpenAI client for modern API calls
# As of May 2025, gpt-4o is a strong general-purpose model.
# The Responses API is also available for agentic workflows.[2]
client = openai.OpenAI()

def chat_with_ai_stateless(user_message: str) -> str:
    """
    A simplified function demonstrating a stateless AI interaction.
    Each call is independent, with no memory of prior messages.
    """
    response = client.chat.completions.create(
        model="gpt-4o", # Updated model to gpt-4o [2]
        messages=[{"role": "user", "content": user_message}]
    )
    return response.choices.message.content

# Every message is treated as a brand new conversation
print(chat_with_ai_stateless("My name is Sarah")) # Expected: "Hello! How can I help you?"
print(chat_with_ai_stateless("What's my name?")) # Expected: "I don't have that information"
```

In the example above, each invocation of chat_with_ai_stateless is an isolated request to the language model. The model receives only the current user_message and has no access to the preceding turn where the user identified themselves. Consequently, it cannot recall the user's name. This highlights a profound limitation: without a mechanism to carry context forward, the agent cannot engage in multi-turn dialogues, personalize interactions, or build a cumulative understanding of the user's needs over time.
The implications of this statelessness extend beyond mere functional limitations. An AI agent that constantly requires users to repeat information or re-establish context incurs significant hidden costs. Users experience frustration, leading to increased abandonment rates and a diminished perception of the agent's utility. Each repeated query, necessitated by the agent's lack of memory, consumes additional computational resources and API tokens, directly inflating operational expenses. Thus, the "expensive" aspect of the "expensive parrot" is multifaceted, encompassing not only direct compute and API costs but also the indirect costs associated with poor user experience and wasted effort. This underscores a crucial point: investing in robust memory management is not merely a feature enhancement but a critical strategy for both cost optimization and user retention in production-grade AI agent deployments.

1.2. Short-Term vs. Long-Term Memory

Effective memory management in AI agents is typically categorized into two primary types, each serving distinct purposes in maintaining conversational coherence and fostering deeper user engagement.
Short-Term Memory is analogous to an AI's "conversation notebook". Its function is to retain context within a single, ongoing conversation. This type of memory is crucial for enabling basic conversational coherence, allowing the agent to understand follow-up questions and refer back to previously mentioned details within the same session. However, this memory is ephemeral; it is "erased when the conversation ends".
Long-Term Memory, conversely, functions like a "personal filing cabinet" for the AI. This enables the agent to remember users across different sessions, learn their preferences, recall past interactions, and build a cumulative relationship over time. Long-term memory is vital for personalization, allowing the AI to provide tailored responses and proactively anticipate user needs based on historical data.
The following table provides an overview of how these memory types, along with advanced and collaborative memory patterns, are implemented across the leading AI frameworks discussed in this report:

Memory Type	Description	LangChain Implementation	Pydantic AI Implementation	Agno Implementation	Key Benefit
Short-Term	Context within	ConversationB	Structured	Built-in	Coherent,

Memory	a single conversation, ephemeral.	ufferMemory, ConversationSummaryBufferMemory, ConversationBufferWindowMemory (via RunnableWithMessageHistory or LangGraph)	message_history management, custom optimization strategies	conversational context for Agent	natural single-session dialogues
Long-Term Memory	Remembers users across sessions, learns preferences, durable.	ConversationEntityMemory (persisted externally), ConversationKGMemory	VectorMemorySystem with topic extraction, importance scoring (conceptual vector DB)	VectorMemory with user-specific collections, RAG	Personalized, evolving user interactions, knowledge retention
Advanced Memory	Beyond text: multi-modal patterns, intelligent compression.	CombinedMemory (Summary, Entity, KG), custom user pattern tracking	Custom history_processors, importance scoring, topic extraction	Built-in vector storage, sophisticated retrieval, Agent introspection	Deeper user understanding, cost efficiency, optimized recall
Team Memory	Multiple agents sharing knowledge, collaboration, conflict resolution.	SharedMemoryStore (conceptual multi-agent architecture)	Not explicitly detailed, but multi-agent workflows are supported	SharedMemory, Crew for multi-agent orchestration, built-in sync	Collective intelligence, specialized expertise, consistent responses

This table serves as a roadmap, illustrating how different frameworks approach the critical challenge of equipping AI agents with memory, from basic conversational recall to sophisticated, collaborative knowledge retention.

2. Short-Term Memory Implementations

Short-term memory is fundamental for any conversational AI agent, ensuring that interactions within a single session remain coherent and contextually relevant. This section explores how leading frameworks, LangChain and Pydantic AI, implement and manage this crucial aspect of AI memory.

2.1. LangChain for Conversational Context

LangChain has undergone significant architectural evolution, moving from monolithic "Chains" to more modular "Runnables" through its Expression Language (LCEL).[3] While older memory classes like
ConversationBufferMemory remain available for backward compatibility, their integration into modern LCEL chains is primarily facilitated by RunnableWithMessageHistory and ChatMessageHistory.[3] This shift emphasizes explicit input/output handling and better integration with asynchronous programming patterns.

2.1.1. Basic Conversation Memory (Updated LCEL Approach)

The simplest form of short-term memory involves remembering every message exchanged within a conversation. In LangChain, this is traditionally handled by ConversationBufferMemory. The updated approach leverages RunnableWithMessageHistory to seamlessly integrate this memory into LCEL chains.

Python

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import HumanMessage, AIMessage
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_openai import ChatOpenAI # Updated to use ChatOpenAI for chat models [7,
8]
from langchain.memory import ConversationBufferMemory # Still used for its internal
chat_memory
from langchain_core.chat_history import BaseChatMessageHistory
from typing import Dict, List
import uuid
import asyncio

# Initialize ChatOpenAI model (using gpt-4o as a modern alternative)
llm = ChatOpenAI(model="gpt-4o", temperature=0.7) # gpt-4o is a strong general-purpose
model as of May 2025 [2]

# Define the prompt template with a placeholder for chat history
```

```
# The 'chat_history' variable name must match the memory_key in ConversationBufferMemory
prompt = ChatPromptTemplate.from_messages(
    [
        MessagesPlaceholder(variable_name="chat_history"), # Essential for LCEL memory
integration [3]
        ("human", "{input}"),
    ]
)

# Create a simple chain with the prompt and LLM
chain = prompt | llm

# In-memory store for session histories (for demonstration purposes only)
# In a production environment, this would be a persistent store like Redis or a database
store: Dict = {}

def get_session_history(session_id: str) -> BaseChatMessageHistory:
    """
    Retrieves or creates a chat history for a given session ID.
    ConversationBufferMemory uses InMemoryChatMessageHistory by default.
    """
    if session_id not in store:
        store[session_id] = ConversationBufferMemory(
            memory_key="chat_history", # Must match MessagesPlaceholder variable_name
            return_messages=True # Essential for use with ChatPromptTemplate
        ).chat_memory
    return store[session_id]

# Wrap the chain with RunnableWithMessageHistory
# This component automatically handles loading and saving chat history for each invocation.
with_message_history = RunnableWithMessageHistory(
    chain,
    get_session_history,
    input_messages_key="input", # Key for the new human input in the chain's input dictionary
    history_messages_key="chat_history", # Key for the chat history in the prompt
)

async def run_conversation():
    session_id = str(uuid.uuid4()) # Generate a unique session ID for each conversation
    config = {"configurable": {"session_id": session_id}} # Configuration for
RunnableWithMessageHistory [3]

    print(f"Starting conversation for session: {session_id}")
```

```
    # The AI now remembers context within this session!
    response1 = await with_message_history.invoke(
        {"input": "Hi, I'm Sarah and I love hiking"},
        config=config
    )
    print(f"AI: {response1.content}")

    response2 = await with_message_history.invoke(
        {"input": "What outdoor activities would you recommend for me?"},
        config=config
    )
    print(f"AI: {response2.content}")

    # Demonstrate starting a new conversation (new session_id)
    new_session_id = str(uuid.uuid4())
    new_config = {"configurable": {"session_id": new_session_id}}
    print(f"\nStarting new conversation for session: {new_session_id}")
    response3 = await with_message_history.invoke(
        {"input": "Hello, who am I?"},
        config=new_config
    )
    print(f"AI: {response3.content}") # The AI will not remember Sarah from the previous
session

# To run this example:
# asyncio.run(run_conversation())
```

This updated code illustrates LangChain's modern conversational memory pattern using LCEL. Instead of the older ConversationChain, a ChatPromptTemplate is constructed with a MessagesPlaceholder to explicitly define where the conversation history should be injected. RunnableWithMessageHistory then acts as an orchestrator, automatically managing the ChatMessageHistory (provided by the get_session_history function) by fetching relevant history for each invocation, injecting it into the prompt, and saving new messages after the LLM's response.[3] This approach represents the idiomatic way to handle conversational memory in contemporary LangChain applications.

This architectural evolution within LangChain, from monolithic "Chains" to modular "Runnables," is a significant development. The older memory classes and ConversationChain are now considered legacy, with a strong recommendation to migrate to RunnableWithMessageHistory or LangGraph for managing chat history within LCEL chains.[3] This indicates a fundamental shift towards prioritizing flexibility, explicit input/output handling, and better integration with asynchronous patterns and streaming. The memory_key and MessagesPlaceholder become central to how memory is injected into prompts. For developers, this means adapting to new patterns that emphasize building custom chains of runnables and explicitly managing how memory feeds into prompts, rather than relying on the implicit behaviors of older ConversationChain objects. This also suggests a move towards more explicit state management, even as RunnableWithMessageHistory abstracts some of the underlying complexities.

2.1.2. Smart Memory with Summaries (Updated LCEL Approach)

For longer conversations, simply buffering all messages can lead to excessive token usage, increased latency, and higher costs. To mitigate this, "smart memory" strategies summarize older parts of the conversation while retaining recent messages in full detail. LangChain's ConversationSummaryBufferMemory is designed for this purpose.

Python

```python
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import HumanMessage, AIMessage
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_openai import ChatOpenAI # Updated to ChatOpenAI [7, 8]
from langchain.memory import ConversationSummaryBufferMemory # Preferred for LCEL
integration [5]
from langchain_core.chat_history import BaseChatMessageHistory
from typing import Dict, List
import uuid
import asyncio

# Initialize ChatOpenAI model
llm = ChatOpenAI(model="gpt-4o", temperature=0.7) # Using gpt-4o [2]

# Define the prompt template
prompt = ChatPromptTemplate.from_messages(
    [
        MessagesPlaceholder(variable_name="chat_history"),
        ("human", "{input}"),
    ]
)

# Create a chain
chain = prompt | llm

# In-memory store for session histories
```

```
store: Dict = {}

def get_session_history_summary(session_id: str) -> BaseChatMessageHistory:
    """
    Retrieves or creates a chat history that summarizes older messages.
    ConversationSummaryBufferMemory summarizes older messages while keeping recent
ones verbatim.
    It requires an LLM for summarization.
    """
    if session_id not in store:
        store[session_id] = ConversationSummaryBufferMemory(
            llm=llm, # The LLM used for summarization [5, 9]
            max_token_limit=1000, # Summarize when the conversation history hits this token limit
[1, 5]
            memory_key="chat_history",
            return_messages=True
        ).chat_memory # Access the underlying ChatMessageHistory
    return store[session_id]

# Wrap the chain with RunnableWithMessageHistory
with_summary_history = RunnableWithMessageHistory(
    chain,
    get_session_history_summary,
    input_messages_key="input",
    history_messages_key="chat_history",
)

async def run_summary_conversation():
    session_id = str(uuid.uuid4())
    config = {"configurable": {"session_id": session_id}}
    print(f"Starting summary conversation for session: {session_id} (max_token_limit=1000)")

    print("User: Tell me about the history of artificial intelligence, starting from its early
concepts.")
    response1 = await with_summary_history.invoke(
        {"input": "Tell me about the history of artificial intelligence, starting from its early
concepts."},
        config=config
    )
    print(f"AI: {response1.content[:150]}...") # Truncate for display

    print("\nUser: What about its ethical implications and the challenges of bias in AI systems?")
    response2 = await with_summary_history.invoke(
```

```
        {"input": "What about its ethical implications and the challenges of bias in AI systems?"},
        config=config
    )
    print(f"AI: {response2.content[:150]}...")

    # Simulate a long conversation to trigger summarization
    print("\nSimulating a long conversation to trigger summarization...")
    long_messages =
    for i, msg in enumerate(long_messages):
        print(f"User (Turn {i+3}): {msg}")
        await with_summary_history.invoke(
            {"input": msg},
            config=config
        )
        print(f"AI (Turn {i+3}):...")
        await asyncio.sleep(0.5) # Simulate delay

    print("\nUser: Can you summarize our discussion so far?")
    response_summary = await with_summary_history.invoke(
        {"input": "Can you summarize our discussion so far?"},
        config=config
    )
    print(f"AI: {response_summary.content}")
    # The AI will automatically summarize older parts of the conversation
    # while keeping recent messages in full detail, demonstrating token management.

# To run this example:
# asyncio.run(run_summary_conversation())
```

ConversationSummaryBufferMemory is a more robust choice for LCEL integration than its predecessor, ConversationSummaryMemory.[5] It intelligently summarizes older portions of the conversation while preserving recent messages verbatim, effectively managing token limits and reducing API costs. The
max_token_limit parameter dictates the threshold at which summarization occurs. This approach is particularly critical for long-running conversations, as it prevents prompt overflow and significantly reduces the computational and financial overhead associated with processing extensive chat histories.

The use of summarization introduces a nuanced consideration regarding the balance between cost, performance, and conversational depth. While ConversationSummaryBufferMemory helps manage token costs and prompt length, thereby potentially improving response latency, the summarization process itself consumes LLM tokens and introduces a slight delay. This highlights a fundamental trade-off inherent in memory strategies. The choice between a full buffer, a summarized history, or a sliding window (as discussed next) is not arbitrary; it represents a strategic decision that balances recall accuracy, token expenditure, and response latency. Over-summarization risks losing critical details, while neglecting summarization leads to escalating costs and slower interactions. Developers must therefore carefully profile and evaluate different memory strategies based on their specific application's requirements for conversational depth, user experience, and operational budget. This is not a one-size-fits-all solution but a spectrum of choices, each with distinct performance and cost profiles.

2.1.3. Sliding Window Memory (Updated LCEL Approach)

Another effective strategy for managing short-term memory is the "sliding window" approach. This method retains only the most recent N messages, discarding older ones. It is often considered the "sweet spot" for many applications, balancing sufficient context retention with efficient resource usage.

Python

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.messages import HumanMessage, AIMessage
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_openai import ChatOpenAI # Updated to ChatOpenAI [7, 8]
from langchain.memory import ConversationBufferWindowMemory # Used for sliding window
memory
from langchain_core.chat_history import BaseChatMessageHistory
from typing import Dict, List
import uuid
import asyncio

# Initialize ChatOpenAI model
llm = ChatOpenAI(model="gpt-4o", temperature=0.7) # Using gpt-4o [2]

# Define the prompt template
prompt = ChatPromptTemplate.from_messages(
    [
        MessagesPlaceholder(variable_name="chat_history"),
        ("human", "{input}"),
    ]
)

# Create a chain
chain = prompt | llm
```

```python
# In-memory store for session histories
store: Dict = {}

def get_session_history_window(session_id: str) -> BaseChatMessageHistory:
    """
    Retrieves or creates a chat history that keeps only the last 'k' messages.
    'k' refers to the total number of messages (human and AI turns combined).
    """
    if session_id not in store:
        store[session_id] = ConversationBufferWindowMemory(
            k=10, # Keeps the last 10 messages (e.g., 5 user-AI turns) [1, 6]
            memory_key="chat_history",
            return_messages=True
        ).chat_memory
    return store[session_id]

# Wrap the chain with RunnableWithMessageHistory
with_window_history = RunnableWithMessageHistory(
    chain,
    get_session_history_window,
    input_messages_key="input",
    history_messages_key="chat_history",
)

async def run_window_conversation():
    session_id = str(uuid.uuid4())
    config = {"configurable": {"session_id": session_id}}
    print(f"Starting window conversation for session: {session_id} (k=10 messages)")

    messages =

    for i, msg in enumerate(messages):
        print(f"\nUser: {msg}")
        response = await with_window_history.invoke({"input": msg}, config=config)
        print(f"AI: {response.content[:100]}...") # Truncate for display
        # In a real scenario, you'd observe the 'chat_history' in the LangSmith trace to see the
window effect.
        await asyncio.sleep(0.1)

    print("\n--- After 10+ messages, the window memory is active ---")
    print("User: What was the first thing I told you?")
    response_forgotten = await with_window_history.invoke(
```

```
        {"input": "What was the first thing I told you?"},
        config=config
    )
    print(f"AI: {response_forgotten.content}") # Should not remember "Hi, I'm Alice." if k=10 was
exceeded

# To run this example:
# asyncio.run(run_window_conversation())
```

ConversationBufferWindowMemory maintains a fixed-size window of the most recent messages. This component is a pragmatic choice for many applications, as it strikes a balance between retaining sufficient conversational context and managing computational costs and performance. It avoids the overhead associated with summarization while ensuring that the immediate conversational flow remains coherent for a reasonable duration.

2.2. Pydantic AI for Structured Conversation History

Pydantic AI is a Python agent framework designed for building production-grade applications with generative AI, emphasizing clean, structured data and type safety.[10] It provides developers with fine-grained control over conversation history, enabling the implementation of custom memory optimization strategies.

2.2.1. Structured Memory Management

Pydantic AI's approach to memory management centers around its Agent class and the explicit handling of message_history as a list of ModelMessage objects (e.g., UserMessage, AssistantMessage, SystemMessage).[12] This structured approach allows for precise control over the conversation context.

Python

```
from pydantic_ai import Agent
from pydantic_ai.messages import ModelMessage, SystemMessage, UserMessage,
AssistantMessage # Added AssistantMessage for completeness
from typing import List, Dict, Any
import asyncio
from datetime import datetime # Can be used for custom logic like timestamping

class ConversationManager:
```

```
"""
Manages conversation history using Pydantic AI's structured message types.
"""
def __init__(self):
    # Initialize Agent with a specific model. gpt-4o is a good modern choice.
    self.agent = Agent(
        'openai:gpt-4o', # Updated model to gpt-4o [11, 12, 13]
        system_prompt='You are a helpful assistant with perfect memory of our conversation.'
    )
    # conversation_history should be a list of ModelMessage, storing the full dialogue.
    self.conversation_history: List[ModelMessage] =

async def chat(self, user_input: str) -> str:
    """
    Processes a user message, gets an AI response, and updates the conversation history.
    """
    # Add the user's message to our history
    user_message = UserMessage(content=user_input)
    self.conversation_history.append(user_message)

    # Get AI response with full conversation context
    # The `run` method takes `message_history` to provide context to the LLM [12]
    result = await self.agent.run(
        message_history=self.conversation_history # Pass the full history
    )

    # Add AI's response to history
    # result.new_messages() returns messages generated during the current run [12]
    self.conversation_history.extend(result.new_messages())

    # Return the content of the assistant's primary response
    if result.output:
        return result.output
    return "No response generated."

def get_conversation_summary(self) -> Dict[str, Any]:
    """
    Provides a conceptual summary of the current conversation.
    In a real system, this might involve another Pydantic AI agent for summarization.
    """
    return {
        "total_messages": len(self.conversation_history),
        "user_messages": len([msg for msg in self.conversation_history if isinstance(msg,
```

```
UserMessage)]),
            "assistant_messages": len([msg for msg in self.conversation_history if isinstance(msg,
AssistantMessage)]),
            "recent_topics": self._extract_recent_topics() # Placeholder for actual topic extraction
        }

    def _extract_recent_topics(self) -> List[str]:
        # This method is a placeholder. In a production system, you might use
        # another Pydantic AI agent to extract topics from self.conversation_history.
        return ["camping", "packing advice"] # Example based on expected usage

# Usage example
async def run_pydantic_ai_chat():
    # Create a conversation manager
    chat_manager = ConversationManager()

    # Have a conversation with memory
    response1 = await chat_manager.chat("Hi, I'm planning a camping trip")
    print(f"AI: {response1}")

    response2 = await chat_manager.chat("What should I pack?")
    print(f"AI: {response2}") # The AI remembers the camping trip context

    # Check conversation statistics
    print("\nConversation Summary:")
    print(chat_manager.get_conversation_summary())

# To run this example:
# asyncio.run(run_pydantic_ai_chat())
```

In this implementation, the Pydantic AI Agent is initialized with a specified model and a system prompt. The chat method incrementally builds the conversation_history list by appending UserMessage objects and then passing this entire list to agent.run(). The result.new_messages() method captures the AI's responses from the current turn, which are then appended back to the conversation_history.[12] This explicit management of the message_history provides developers with direct and granular control over the conversation context, which is particularly advantageous for implementing sophisticated custom optimization strategies.

2.2.2. Custom Memory Optimization

Pydantic AI's design facilitates custom memory optimization by allowing developers to extend core classes or utilize specific hooks. The example below demonstrates a custom history trimming logic, effectively implementing a sliding window for conversation history.

Python

```
from pydantic_ai import Agent
from pydantic_ai.messages import ModelMessage, UserMessage, AssistantMessage,
SystemMessage
from typing import List, Dict, Any
import asyncio

# Re-using ConversationManager from the previous snippet for inheritance demonstration
class ConversationManager:
  def __init__(self):
    self.agent = Agent(
      'openai:gpt-4o',
      system_prompt='You are a helpful assistant with perfect memory of our conversation.'
    )
    self.conversation_history: List[ModelMessage] =

  async def chat(self, user_input: str) -> str:
    user_message = UserMessage(content=user_input)
    self.conversation_history.append(user_message)
    result = await self.agent.run(
      message_history=self.conversation_history
    )
    self.conversation_history.extend(result.new_messages())
    if result.output:
      return result.output
    return "No response generated."

  def get_conversation_summary(self) -> Dict[str, Any]:
    return {
      "total_messages": len(self.conversation_history),
      "user_messages": len([msg for msg in self.conversation_history if isinstance(msg,
UserMessage)]),
      "assistant_messages": len([msg for msg in self.conversation_history if isinstance(msg,
AssistantMessage)]),
      "recent_topics": self._extract_recent_topics()
    }

  def _extract_recent_topics(self) -> List[str]:
```

```
        return ["camping", "packing advice"] # Example

class OptimizedConversationManager(ConversationManager):
    """
    Extends ConversationManager to implement custom history trimming logic.
    """
    def __init__(self, max_messages: int = 20):
        super().__init__()
        self.max_messages = max_messages # Defines the maximum number of messages to
retain in history

    def _trim_history(self):
        """
        Keeps the conversation history from getting too long by trimming older messages.
        This implementation prioritizes keeping the initial system prompt and recent
conversational turns.
        """
        if len(self.conversation_history) > self.max_messages:
            # Pydantic AI's `history_processors` is generally a more idiomatic and integrated way
            # to implement this logic directly within the Agent configuration.[12]
            # However, for direct replication of the article's class-based approach:

            # Identify the initial system message, if present, to ensure it's always retained for
context.
            first_system_message = next((msg for msg in self.conversation_history if
isinstance(msg, SystemMessage)), None)

            # Filter out system messages from the main conversation flow to apply trimming logic
only to dialogue.
            conversational_messages =

            # Determine how many conversational messages to keep, reserving space for the
system message.
            num_to_keep = self.max_messages - (1 if first_system_message else 0)

            if len(conversational_messages) > num_to_keep:
                # Retain only the most recent conversational messages.
                recent_messages = conversational_messages[-num_to_keep:]
                # Reconstruct the history with the system message (if any) followed by recent
dialogue.
                self.conversation_history = ([first_system_message] if first_system_message else) +
recent_messages
            # If history is already within limits or too small to trim, no action is taken.
```

```
    async def chat(self, user_input: str) -> str:
        """
        Overrides the parent chat method to apply history optimization after each exchange.
        """
        # Call the parent chat method to get the AI response and update the raw history.
        response = await super().chat(user_input)
        self._trim_history() # Apply the trimming logic to optimize history size
        return response

# Usage example
async def run_optimized_pydantic_ai_chat():
    optimized_chat_manager = OptimizedConversationManager(max_messages=6) # Small
window for demonstration

    print("Starting optimized conversation (max 6 messages in history):")
    messages =

    for i, msg in enumerate(messages):
        print(f"\nUser: {msg}")
        response = await optimized_chat_manager.chat(msg)
        print(f"AI: {response}")
        print(f"Current history length: {len(optimized_chat_manager.conversation_history)}
messages.")
        await asyncio.sleep(0.1)

    print("\nFinal conversation summary:")
    print(optimized_chat_manager.get_conversation_summary())

# To run this example:
# asyncio.run(run_optimized_pydantic_ai_chat())
```

This OptimizedConversationManager class extends the base ConversationManager and overrides its chat method to invoke _trim_history after each interaction. The _trim_history method implements a basic sliding window, ensuring that only a fixed number of recent messages, along with the initial system prompt, are retained. While this class-based override directly replicates the article's structure, Pydantic AI also offers history_processors as a more integrated and flexible mechanism.[12]

history_processors allow developers to define custom functions that modify the message history before it is sent to the language model, providing a powerful means for filtering sensitive information, managing token costs, or implementing other custom memory strategies.

The distinct approaches to framework design, as seen in LangChain's LCEL and Pydantic AI's

Agent with history_processors, highlight differing philosophies in extensibility. LangChain's LCEL encourages developers to compose small, explicit runnables, offering maximum flexibility in constructing custom pipelines. In contrast, Pydantic AI provides a more opinionated Agent abstraction but offers specific hooks like history_processors for targeted customization.[12] This distinction influences the ease with which custom memory strategies can be implemented and integrated. For projects demanding highly granular control over every processing step, LangChain's LCEL might be preferred. Conversely, for faster development of agents with structured inputs/outputs and built-in memory management, Pydantic AI offers a streamlined experience, with history_processors serving as a powerful point of extensibility. The optimal framework choice thus depends on the specific project's requirements and the desired balance between out-of-the-box functionality and deep customization.

3. Long-Term Memory Implementations

While short-term memory ensures conversational coherence within a single session, long-term memory enables AI agents to recall information across disparate interactions, learn user preferences, and build enduring relationships. This section explores how LangChain, Pydantic AI, and Agno approach the implementation of long-term memory.

3.1. LangChain’s Entity Memory for Knowledge Graphs

LangChain provides ConversationEntityMemory as a mechanism for long-term memory, designed to identify and track key entities within conversations and maintain a knowledge graph of facts associated with them.

3.1.1. Remembering What Matters

ConversationEntityMemory leverages an underlying language model to automatically identify and extract important entities (such as people, places, or companies) from the conversation history. It then summarizes and stores facts about these entities, effectively building a dynamic knowledge graph.[1]

Python

```
from langchain.memory import ConversationEntityMemory
from langchain.memory.prompt import ENTITY_MEMORY_CONVERSATION_TEMPLATE
from langchain.chains import ConversationChain # Still functional for this specific memory
```

```
type demonstration
from langchain_openai import ChatOpenAI # Updated to ChatOpenAI [4, 7, 8, 14]
import asyncio

# Initialize ChatOpenAI models for entity extraction and conversation.
# A lower temperature is often preferred for factual consistency during entity extraction.
llm_entity_extraction = ChatOpenAI(model="gpt-4o", temperature=0) # Using gpt-4o [2]
llm_conversation = ChatOpenAI(model="gpt-4o", temperature=0.7) # Using gpt-4o [2]

# Create entity memory that tracks important information.
# Note: ConversationEntityMemory is marked as deprecated but remains functional for
demonstration.
# Its `llm` parameter is crucial for the internal entity extraction and summarization
processes.[14]
entity_memory = ConversationEntityMemory(
    llm=llm_entity_extraction,
    k=10, # Configures memory to remember details about the 10 most recent entities [1, 14]
    # entity_extraction_prompt and entity_summarization_prompt can be customized for
specific needs [14]
)

# Set up the conversation with entity memory using ConversationChain.
# ConversationChain is a legacy component, but it directly integrates with this memory type.
# ENTITY_MEMORY_CONVERSATION_TEMPLATE is a specific prompt designed to work with
ConversationEntityMemory.[4]
conversation = ConversationChain(
    llm=llm_conversation,
    memory=entity_memory,
    prompt=ENTITY_MEMORY_CONVERSATION_TEMPLATE,
    verbose=True # Enables verbose output to observe the memory's internal operations
)

async def run_entity_memory_conversation():
    print("--- LangChain Entity Memory Demonstration ---")
    print("User: Hi, I'm Sarah from TechCorp. I'm working on a project about sustainable energy
with my colleague Mike.")
    response1 = await conversation.ainvoke( # Using ainvoke for asynchronous execution
        {"input": "Hi, I'm Sarah from TechCorp. I'm working on a project about sustainable energy
with my colleague Mike."}
    )
    print(f"AI: {response1['response']}")

    print("\nUser: How's Mike doing with the sustainable energy research?")
```

```
    response2 = await conversation.ainvoke(
        {"input": "How's Mike doing with the sustainable energy research?"}
    )
    print(f"AI: {response2['response']}")
    # The AI demonstrates recall, remembering Sarah works at TechCorp, is collaborating with
Mike,
    # and their focus on sustainable energy.

    print("\nUser: What is TechCorp known for?")
    response3 = await conversation.ainvoke(
        {"input": "What is TechCorp known for?"}
    )
    print(f"AI: {response3['response']}")

    # Inspect entities stored in memory (for debugging and understanding the knowledge
graph)
    print("\n--- Current Entities in Memory ---")
    # Accessing entity_store directly for demonstration. This dictionary holds the extracted
facts about entities.
    print(entity_memory.entity_store)

# To run this example:
# asyncio.run(run_entity_memory_conversation())
```

This example demonstrates how ConversationEntityMemory utilizes an LLM to identify and summarize entities from the conversation.[1] The ENTITY_MEMORY_CONVERSATION_TEMPLATE is specifically crafted to integrate this entity information into the prompt, allowing the AI to track and leverage facts about specific individuals, organizations, or concepts mentioned. This capability provides a foundational form of knowledge graph, significantly enhancing the agent's long-term contextual understanding.

3.1.2. Making Entity Memory Persistent

For long-term memory to truly function across different sessions, the extracted entity knowledge graph must be saved to a persistent storage medium.[1] This allows the AI to recall information about users or topics even after the original conversation has concluded.

Python

```
import json
```

```
from typing import Dict, Any
from datetime import datetime
from langchain.memory import ConversationEntityMemory
from langchain_openai import ChatOpenAI # Updated to ChatOpenAI [7, 8]
import asyncio
import os # For file cleanup

class PersistentEntityMemory:
    """
    Manages the persistence of LangChain's ConversationEntityMemory.
    """
    def __init__(self, storage_file: str = "entity_memory.json"):
        self.storage_file = storage_file
        # Initialize ChatOpenAI for entity extraction.
        self.llm = ChatOpenAI(model="gpt-4o", temperature=0) # Using gpt-4o [2]

        # Set up LangChain entity memory.
        # While ConversationEntityMemory is deprecated, it's used here to illustrate persistence
of its entity_store.
        self.langchain_memory = ConversationEntityMemory(
            llm=self.llm,
            k=10 # Number of recent entities to consider [14]
        )

        # Load existing entities into LangChain memory's entity_store upon initialization.
        # The entity_store is a dictionary that holds the extracted facts about entities.
        self.langchain_memory.entity_store = self.load_entities()
        print(f"Loaded {len(self.langchain_memory.entity_store)} entities from {self.storage_file}")

    def load_entities(self) -> Dict[str, Any]:
        """Loads entities from the specified JSON storage file."""
        try:
            with open(self.storage_file, 'r') as f:
                return json.load(f)
        except FileNotFoundError:
            return {}
        except json.JSONDecodeError:
            print(f"Warning: Could not decode JSON from {self.storage_file}. Starting with empty
memory.")
            return {}

    def save_entities(self):
        """Saves the current entities from LangChain memory's entity_store to the storage file."""
```

```
        with open(self.storage_file, 'w') as f:
            # Ensure the entity_store content is JSON serializable.
            json.dump(self.langchain_memory.entity_store, f, indent=2)
        print(f"Saved {len(self.langchain_memory.entity_store)} entities to {self.storage_file}")

    async def remember_user(self, user_id: str, conversation_text: str):
        """
        Processes conversation text to extract and remember information about a user.
        In a full LangChain integration, this would typically happen implicitly via a
ConversationChain.
        Here, we simulate updating a user-specific store for demonstration of persistence.
        """
        # Simulate entity extraction from the conversation_text.
        # In a real application, this would involve a more sophisticated LLM-driven extraction
process
        # or leveraging the ConversationEntityMemory's internal mechanisms by running a
ConversationChain.
        extracted_info = {}
        if "TechCorp" in conversation_text:
            extracted_info["company"] = "TechCorp"
        if "Sarah" in conversation_text:
            extracted_info["name"] = "Sarah"
        if "marketing director" in conversation_text:
            extracted_info["role"] = "marketing director"
        if "sustainable energy" in conversation_text:
            extracted_info["project_focus"] = "sustainable energy"
        if "Mike" in conversation_text:
            extracted_info["colleague"] = "Mike"

        # Store user-specific information within the entity_store.
        # The entity_store is used here as a simple persistent dictionary for user data.
        if user_id not in self.langchain_memory.entity_store:
            self.langchain_memory.entity_store[user_id] = {}

        user_data = self.langchain_memory.entity_store[user_id]
        user_data.update({
            "last_conversation_snippet": conversation_text[:200], # Keep a snippet for quick
context
            "last_seen": datetime.now().isoformat(),
            "conversation_count": user_data.get("conversation_count", 0) + 1,
            "extracted_details": extracted_info # Store the simulated extracted details
        })
```

```
        # Save the updated entities to persistent storage.
        self.save_entities()

    def get_user_context(self, user_id: str) -> str:
        """Retrieves and formats relevant context about a user from persistent memory."""
        user_info = self.langchain_memory.entity_store.get(user_id)
        if not user_info:
            return "This appears to be a new user."

        return f"""
Previous context about this user ({user_id}):
- Last seen: {user_info.get('last_seen', 'Unknown')}
- Conversations: {user_info.get('conversation_count', 0)}
- Last topic: {user_info.get('last_conversation_snippet', 'No previous context')}
- Extracted details: {json.dumps(user_info.get('extracted_details', {}), indent=2)}
"""

# Usage example
async def run_persistent_entity_memory():
    # Clean up previous memory file for a fresh start in the demonstration.
    if os.path.exists("entity_memory.json"):
        os.remove("entity_memory.json")
        print("Cleaned up old entity_memory.json for a fresh demo.")

    memory_system = PersistentEntityMemory()

    # Simulate a user starting a conversation.
    user_id = "sarah_123"
    user_context = memory_system.get_user_context(user_id)
    print(user_context)

    # After a conversation ends, remember new information about the user.
    await memory_system.remember_user(user_id, "Sarah discussed her new role as marketing
director at TechCorp and her interest in sustainable energy.")

    # Retrieve context again to observe the updated information.
    print("\n--- After first interaction ---")
    user_context_updated = memory_system.get_user_context(user_id)
    print(user_context_updated)

    # Simulate another interaction, further updating user information.
    await memory_system.remember_user(user_id, "Sarah asked about Mike's progress on the
sustainable energy project.")
```

```
    print("\n--- After second interaction ---")
    user_context_further_updated = memory_system.get_user_context(user_id)
    print(user_context_further_updated)

# To run this example:
# asyncio.run(run_persistent_entity_memory())
```

This code demonstrates the mechanism for persisting the entity_store, which is the internal dictionary of facts about entities maintained by ConversationEntityMemory, to a JSON file. The load_entities and save_entities methods handle the reading and writing operations, allowing the AI to retain knowledge about entities and their associated facts across different sessions.[1] The
remember_user method simulates the process of updating this persistent store based on new conversation content, and get_user_context retrieves the stored information for subsequent interactions.
This approach to long-term memory reveals a hybrid architectural pattern. LangChain's ConversationEntityMemory leverages the LLM's capabilities for the *extraction* and *summarization* of structured facts from conversational text.[1] However, the persistence example then saves these extracted facts to a separate, traditional data store (in this case, a JSON file, but more realistically a database or vector store). This illustrates that true long-term memory in AI agents often necessitates a multi-component architecture. This architecture combines the intelligent processing capabilities of LLMs for understanding and generating knowledge with the robust storage, indexing, and retrieval efficiencies of traditional data management systems. The system moves beyond merely feeding raw text into prompts, establishing a more robust and queryable knowledge base that augments the LLM's inherent capabilities.

3.2. Pydantic AI with Vector Memory for Semantic Recall

Pydantic AI offers a structured approach to memory management, which can be extended to implement sophisticated long-term memory systems. A key aspect of this is the integration of vector memory for semantic recall, enabling the AI to find conceptually similar information rather than relying solely on exact keyword matches.

3.2.1. Vector Memory System

This system leverages Pydantic models for structured memory entries and the Pydantic AI Agent for intelligent processing such as topic extraction and importance scoring. While the example uses a simplified in-memory list and keyword matching for recall, it conceptually represents a system that would utilize a vector database in a production environment.

Python

```
from pydantic_ai import Agent
from pydantic import BaseModel, Field # Using Field for default values as per Pydantic v2+
from pydantic_ai.messages import SystemMessage, UserMessage # For constructing LLM
prompts
from typing import List, Optional, Dict, Any
import asyncio
import json
from datetime import datetime

class MemoryEntry(BaseModel):
    """
    Defines the structured schema for a single memory entry.
    """
    user_id: str
    content: str
    timestamp: datetime = Field(default_factory=datetime.now) # Automatically sets current
time on creation
    conversation_id: str
    topics: List[str] = Field(default_factory=list) # List of extracted topics
    importance_score: float = Field(default=0.5) # Importance score for the memory

class VectorMemorySystem:
    """
    Manages a conceptual vector memory system for Pydantic AI agents.
    In a production setting, `memory_store` would interface with a vector database.
    """
    def __init__(self):
        self.agent = Agent('openai:gpt-4o') # Updated model to gpt-4o [11, 12]
        self.memory_store: List[MemoryEntry] =
        self.storage_file = 'vector_memory.json'
        self.load_memory()

    def load_memory(self):
        """Loads existing memories from a JSON storage file."""
        try:
            with open(self.storage_file, 'r') as f:
                data = json.load(f)
                # Pydantic v2 `model_validate` for robust parsing of dictionary data into
MemoryEntry objects.
```

```
            self.memory_store = [MemoryEntry.model_validate(entry) for entry in data]
        except FileNotFoundError:
          self.memory_store =
        except json.JSONDecodeError:
          print(f"Warning: Could not decode JSON from {self.storage_file}. Starting with empty
memory.")
          self.memory_store =

    def save_memory(self):
        """Saves current memories to the persistent JSON storage file."""
        with open(self.storage_file, 'w') as f:
          # Pydantic v2 `model_dump(mode='json')` for serialization to JSON compatible format.
          json.dump([entry.model_dump(mode='json') for entry in self.memory_store], f,
indent=2)

    async def store_memory(self, user_id: str, content: str, conversation_id: str):
        """
        Stores a new memory, automatically extracting topics and calculating importance using
the AI agent.
        """
        # Extract topics using the Pydantic AI agent.
        # The agent is prompted to return a comma-separated list of topics.
        topic_prompt = f"Extract 3-5 key topics or themes from this text. Return as
comma-separated list: {content}"
        topic_result = await self.agent.run(topic_prompt) # Uses agent.run for LLM interaction
[12]
        topics = [topic.strip() for topic in topic_result.output.split(',') if topic.strip()] #
Access.output for content

        # Calculate importance using the Pydantic AI agent.
        importance = await self._calculate_importance(content)

        # Create and store the new memory entry.
        memory = MemoryEntry(
          user_id=user_id,
          content=content,
          conversation_id=conversation_id,
          topics=topics,
          importance_score=importance
        )

        self.memory_store.append(memory)
        self.save_memory()
```

```
    async def _calculate_importance(self, content: str) -> float:
        """
        Calculates how important a memory is (0.0 to 1.0) using the AI agent.
        The agent is prompted to return only the numerical score.
        """
        importance_prompt = f"""
Rate the importance of the following text on a scale of 0.0 to 1.0,
where 1.0 is highly important and 0.0 is not important.
Consider keywords like 'problem', 'issue', 'urgent', 'important', 'deadline', 'project'.
Return only the score as a float.
Text: {content}
"""
        # Use the agent to get the importance score.
        score_result = await self.agent.run(importance_prompt)
        try:
            score = float(score_result.output.strip())
            return max(0.0, min(1.0, score)) # Ensure score is within the valid range
        except ValueError:
            return 0.5 # Default importance if parsing fails

    async def recall_memories(self, user_id: str, query: str, limit: int = 5) -> List[MemoryEntry]:
        """
        Finds relevant memories for a given user and query.
        In a production environment, this would involve a vector similarity search.
        For this example, an enhanced keyword matching combined with importance score is
used.
        """
        user_memories = [m for m in self.memory_store if m.user_id == user_id]

        if not user_memories:
            return

        # Simple keyword matching (in production, you'd use vector similarity)
        query_words = set(query.lower().split())
        scored_memories =

        for memory in user_memories:
            memory_words = set(memory.content.lower().split())
            topic_words = set(' '.join(memory.topics).lower().split())

            # Calculate relevance score based on keyword and topic overlap, and importance
score.
```

```
            keyword_overlap = len(query_words.intersection(memory_words))
            topic_overlap = len(query_words.intersection(topic_words))

            relevance_score = (keyword_overlap * 0.7) + (topic_overlap * 0.3) +
memory.importance_score

            if relevance_score > 0: # Only include memories with some calculated relevance
                scored_memories.append((relevance_score, memory))

        # Sort by relevance score in descending order and return the top results.
        scored_memories.sort(key=lambda x: x, reverse=True) # Sort by score (x)
        return [memory for _, memory in scored_memories[:limit]]

    async def chat_with_memory(self, user_id: str, message: str, conversation_id: str) -> str:
        """
        Engages in a chat interaction, recalling relevant memories and incorporating them into
the context.
        """
        # Recall relevant memories based on the current user and message.
        relevant_memories = await self.recall_memories(user_id, message)

        # Build context from the recalled memories.
        memory_context = ""
        if relevant_memories:
            memory_context = "Previous context:\n"
            for memory in relevant_memories:
                memory_context += f"- {memory.timestamp.strftime('%Y-%m-%d %H:%M')}:
{memory.content[:100]}...\n"
        else:
            memory_context = "No relevant previous context found."

        # Generate response with the constructed context.
        # Pydantic AI's `message_history` is used to pass the context to the LLM.
        messages_for_llm: List[ModelMessage] =
        if memory_context:
            messages_for_llm.append(SystemMessage(content=memory_context)) # Inject context
as a system message
        messages_for_llm.append(UserMessage(content=message)) # Add the current user
message

        response_result = await self.agent.run(
            message_history=messages_for_llm # Pass the constructed history to the agent
        )
```

```
        ai_response_content = response_result.output

        # Store this interaction (user query + AI response) as a new memory.
        await self.store_memory(
            user_id=user_id,
            content=f"User: {message}\nAssistant: {ai_response_content}",
            conversation_id=conversation_id
        )

        return ai_response_content

# Usage example
async def run_pydantic_ai_vector_memory():
    # Clean up previous memory file for a fresh start in the demonstration.
    import os
    if os.path.exists("vector_memory.json"):
        os.remove("vector_memory.json")
        print("Cleaned up old vector_memory.json for a fresh demo.")

    memory_system = VectorMemorySystem()

    # First conversation for Sarah.
    print("\n--- First Conversation (conv_001) for Sarah ---")
    response1 = await memory_system.chat_with_memory(
        user_id="sarah_123",
        message="I'm working on a machine learning project for my company TechCorp, focusing
on predictive analytics.",
        conversation_id="conv_001"
    )
    print(f"AI: {response1}")
    await asyncio.sleep(0.5) # Simulate time passing

    # Later conversation (different session) for Sarah, related to previous topic.
    print("\n--- Second Conversation (conv_002) for Sarah ---")
    response2 = await memory_system.chat_with_memory(
        user_id="sarah_123",
        message="How's my ML project coming along? I need to present the predictive analytics
results soon.",
        conversation_id="conv_002"
    )
    print(f"AI: {response2}")
    # The system should recall Sarah works at TechCorp on an ML project with predictive
analytics!
```

```
    await asyncio.sleep(0.5)

    # A new user, with a completely different context.
    print("\n--- Third Conversation (conv_003) for John ---")
    response3 = await memory_system.chat_with_memory(
        user_id="john_456",
        message="I'm interested in learning about quantum computing.",
        conversation_id="conv_003"
    )
    print(f"AI: {response3}")
    await asyncio.sleep(0.5)

    # Sarah asks something new, but still within her general domain, to see if memory is
recalled.
    print("\n--- Fourth Conversation (conv_004) for Sarah ---")
    response4 = await memory_system.chat_with_memory(
        user_id="sarah_123",
        message="What are some common challenges in deploying ML models?",
        conversation_id="conv_004"
    )
    print(f"AI: {response4}") # Should recall previous ML project context due to semantic
similarity

    print("\n--- Current Memory Store (for inspection) ---")
    for i, mem in enumerate(memory_system.memory_store):
        print(f"Memory {i+1}: User={mem.user_id}, Topics={mem.topics},
Importance={mem.importance_score:.2f}, Content='{mem.content[:50]}...'")

# To run this example:
# asyncio.run(run_pydantic_ai_vector_memory())
```

This system utilizes Pydantic's BaseModel to define a structured MemoryEntry, ensuring data consistency and type safety. It intelligently leverages the Pydantic AI Agent to perform two critical functions on the memory content: automatic topic extraction and importance scoring. This represents a significant advancement, where the language model itself acts as a dynamic, intelligent processor of memory, enriching it with valuable metadata (topics, importance) that can then be used for more effective retrieval. While the recall_memories method in this demonstration employs a simplified keyword matching approach, it explicitly notes that a vector database would be used in a production environment to facilitate true semantic recall, allowing the system to find conceptually similar information rather than just exact word matches. The chat_with_memory function orchestrates the entire long-term memory loop: it recalls relevant memories, constructs a contextual prompt, generates a response using the agent, and then stores the new interaction (user query + AI response) as a

fresh memory.
This approach highlights the emergence of AI-native data processing for memory. The system does not merely store raw textual data; instead, it actively uses an LLM to *process* the memory content, extracting valuable metadata like topics and importance scores. This marks a notable departure from traditional data processing paradigms, where such enrichment would typically rely on predefined rules or complex, separate NLP pipelines. Here, the LLM itself becomes a dynamic, intelligent engine for structuring, indexing, and managing the memory, enabling more nuanced and effective retrieval. This trend suggests that future memory systems for AI agents will increasingly rely on LLMs not just for interaction, but also for the intelligent organization and management of the memory itself. This could lead to the development of more sophisticated retrieval-augmented generation (RAG) systems where the "R" (retrieval) component is also powered by AI, moving beyond simple keyword or vector similarity to achieve more context-aware and semantically rich memory recall.

3.3. Agno: Production-Grade Memory Architecture

Agno is an open-source framework specifically designed for building production-grade AI agents, emphasizing clean, composable, and Pythonic architectures with built-in support for tools, memory, and reasoning capabilities.[15] It provides robust solutions for memory management, particularly for long-term recall and multi-agent collaboration.

3.3.1. Production Memory Agent

Agno's design integrates vector storage directly into its memory components, abstracting away the complexities of interacting with underlying vector databases. This makes it a strong candidate for production environments requiring scalable and sophisticated memory solutions.[1]

Python

```
from agno import Agent
from agno.memory import VectorMemory # Agno's built-in vector memory component
from agno.models.openai import OpenAIChat # Agno's wrapper for OpenAI chat models [13]
import asyncio
from typing import Dict, List, Any
import json # For formatting memory summary output

class ProductionMemoryAgent:
    """
```

```
    An Agno-based AI agent designed for production environments with integrated long-term
memory.
    Agno automatically handles memory retrieval and storage, leveraging vector capabilities.
    """
    def __init__(self, user_id: str):
        self.user_id = user_id

        # Create user-specific vector memory using Agno's VectorMemory.
        # Agno handles the underlying vector storage (e.g., PgVector as mentioned in its
documentation [15]).
        self.memory = VectorMemory(
            collection_name=f"user_{user_id}_memories", # Unique collection for each user to
isolate memories
            embeddings_model="text-embedding-3-large", # OpenAI's latest embedding model
for high-quality embeddings
            distance_metric="cosine" # Common distance metric for vector similarity
        )

        # Create the Agno agent with integrated memory.
        # Use Agno's OpenAIChat wrapper for the model.
        self.agent = Agent(
            model=OpenAIChat(id="gpt-4o", temperature=0.7), # Updated model to gpt-4o [13]
            memory=self.memory, # Integrate Agno's VectorMemory directly into the agent
            instructions=f"""
            You are an AI assistant with perfect memory of your conversations with this user
({user_id}).
            Always reference relevant past conversations when appropriate.
            Be personal and build on previous interactions.
            """
        )

    async def chat(self, message: str) -> str:
        """
        Engages in a chat interaction. Agno's agent.run() automatically uses the configured
        memory for Retrieval-Augmented Generation (RAG) and storage of new interactions.[13]
        """
        response = await self.agent.run(message)
        return response.output # Agno's run method returns a RunResult object; access its
output content.

    async def get_memory_summary(self) -> Dict[str, Any]:
        """
        Retrieves insights and a summary of the memories stored for this agent.
```

```
        This is a conceptual example demonstrating memory introspection.
        """
        # Agno's memory.search() can retrieve memories. An empty query retrieves all memories
(up to limit).
        memories = await self.memory.search(
            query="",
            limit=100, # Retrieve a reasonable number of memories for summary
            include_metadata=True # Essential to get timestamp and other metadata
        )

        # Sort memories by timestamp to identify the first and last interaction.
        sorted_memories = sorted(memories, key=lambda m: m['metadata'].get('timestamp', ''))

        first_interaction = sorted_memories['metadata']['timestamp'] if sorted_memories else
None
        last_interaction = sorted_memories[-1]['metadata']['timestamp'] if sorted_memories else
None

        return {
            "total_memories": len(memories),
            "conversation_topics": await self._extract_topics(memories),
            "first_interaction": first_interaction,
            "last_interaction": last_interaction
        }

    async def _extract_topics(self, memories: List]) -> List[str]:
        """
        Extracts common topics from a list of memories using the agent's LLM capabilities.
        """
        if not memories:
            return

        # Combine content from recent memories for summarization.
        recent_content = " ".join([mem["content"] for mem in memories[:20]]) # Use content from
the first 20 memories.

        # Use the agent to extract topics by prompting it.
        topic_response = await self.agent.run(
            f"Extract the top 5 key topics discussed in these conversations. Return as a
comma-separated list: {recent_content}"
        )
        return [topic.strip() for topic in topic_response.output.split(',') if topic.strip()]
```

```
# Usage for production
async def main_agno_demo():
    # Create memory-enabled agents for different users.
    sarah_agent = ProductionMemoryAgent("sarah_123")
    john_agent = ProductionMemoryAgent("john_456")

    print("--- Agno Production Memory Demonstration ---")

    # Sarah's conversation.
    print("\nSarah's first interaction:")
    sarah_response = await sarah_agent.chat("I'm launching a new product at TechCorp next
month, it's an AI-powered analytics platform.")
    print(f"Sarah's Agent: {sarah_response}")
    await asyncio.sleep(0.5)

    # John's separate conversation.
    print("\nJohn's first interaction:")
    john_response = await john_agent.chat("I need help with my marketing strategy for a new
e-commerce site.")
    print(f"John's Agent: {john_response}")
    await asyncio.sleep(0.5)

    # Later - Sarah returns, expecting memory recall.
    print("\nSarah's second interaction:")
    sarah_response2 = await sarah_agent.chat("How should I prepare for the product launch,
specifically for the AI analytics platform?")
    print(f"Sarah's Agent (later): {sarah_response2}") # This response should reference
TechCorp and the product launch.

    # Check Sarah's memory summary to see retained information.
    print("\n--- Sarah's Memory Summary ---")
    summary = await sarah_agent.get_memory_summary()
    print(json.dumps(summary, indent=2))

# To run this example:
# asyncio.run(main_agno_demo())
```

Agno's Agent class directly integrates VectorMemory, effectively abstracting away the complexities of vector database interactions for both Retrieval-Augmented Generation (RAG) and memory storage.[1] When the
chat method is invoked, Agno automatically leverages this configured memory to retrieve relevant past information and store new interactions. The get_memory_summary and _extract_topics methods further demonstrate how the agent can introspect its own memory,

providing valuable insights into the stored information and its evolution. Agno's architectural design is geared towards production readiness, offering built-in solutions for scalable memory management, which is a significant advantage for developers building complex AI systems. Comparing Agno and Pydantic AI's approaches to "production-grade" memory reveals distinct philosophies. While Pydantic AI provides the tools for structured memory management and intelligent processing, Agno offers a higher-level abstraction by directly embedding VectorMemory into its Agent class, handling embeddings, vector search, and persistence.[1] This suggests that Agno aims to be a more comprehensive, batteries-included framework for building agents, particularly for teams, by providing opinionated, built-in solutions for common needs like vector memory and multi-agent collaboration. For developers prioritizing rapid deployment and robust out-of-the-box features for complex multi-agent systems, frameworks like Agno offer significant advantages by abstracting away much of the underlying infrastructure, such as vector databases. This allows teams to focus more on the core agent logic and less on the intricate plumbing, thereby accelerating the time to production for sophisticated AI applications.

4. Advanced Memory Patterns and Architectures

Beyond basic short-term and long-term memory, advanced memory patterns enable AI agents to achieve higher levels of intelligence, personalization, and efficiency. These include multi-modal memory, intelligent compression, and collaborative memory for multi-agent systems.

4.1. Multi-Modal Memory and User Pattern Tracking

Traditional AI agent memory often focuses solely on textual conversation history. However, truly intelligent agents benefit from multi-modal memory, which extends beyond text to include user behavior patterns, interaction styles, time preferences, and even emotional context. LangChain provides mechanisms to combine different memory types to achieve sophisticated architectures.

4.1.1. LangChain's Advanced Memory Combinations (Multi-Modal Memory)

LangChain's CombinedMemory allows for the aggregation of various memory components, such as ConversationSummaryMemory, ConversationEntityMemory, and ConversationKGMemory, to build a richer contextual understanding.[1] This example also introduces custom logic for tracking user interaction patterns, which represents a form of non-textual, multi-modal memory.

Python

```
from langchain.memory import (
    CombinedMemory,
    ConversationSummaryMemory,
    ConversationEntityMemory,
    ConversationKGMemory # Knowledge Graph Memory
)
from langchain.chains import ConversationChain # Still used for combining memories directly
for demonstration
from langchain_openai import ChatOpenAI # Updated to ChatOpenAI [7, 8]
from typing import Dict, Any, List
import asyncio
import json # For printing user patterns

class AdvancedMemorySystem:
    """
    Demonstrates combining multiple LangChain memory types with custom user pattern
tracking
    to achieve a form of multi-modal memory.
    """
    def __init__(self):
        # Initialize ChatOpenAI model for all memory components.
        self.llm = ChatOpenAI(model="gpt-4o", temperature=0.7) # Using gpt-4o [2]

        # Different types of memory working together.
        # Note: These memory classes are marked as deprecated but remain functional for
demonstrating combination.
        self.summary_memory = ConversationSummaryMemory(
            llm=self.llm,
            max_token_limit=1000 # Summarizes conversation to manage token limits
        )

        self.entity_memory = ConversationEntityMemory(
            llm=self.llm,
            k=15 # Tracks details about the 15 most recent entities
        )

        # Knowledge graph memory - tracks relationships between concepts.
        self.kg_memory = ConversationKGMemory(
            llm=self.llm,
            k=10 # Number of knowledge graph triples to store
```

```
        )

        # Combine all memory types using CombinedMemory.[17]
        self.combined_memory = CombinedMemory(
            memories=[
                self.summary_memory,
                self.entity_memory,
                self.kg_memory
            ]
        )

        # Custom component for tracking user patterns (not part of LangChain's core memory
classes).
        # This represents a form of multi-modal memory, capturing non-textual interaction data.
        self.user_patterns: Dict] = {}

    def track_interaction_pattern(self, user_id: str, interaction_data: Dict[str, Any]):
        """
        Tracks and updates patterns in how users interact, such as preferred response length,
        typical session duration, common topics, and complexity preference.
        """
        if user_id not in self.user_patterns:
            self.user_patterns[user_id] = {
                'preferred_response_length': 'medium',
                'typical_session_duration': 0.0, # in minutes
                'common_topics':,
                'interaction_times':, # e.g., list of hours of interaction
                'complexity_preference': 'intermediate'
            }

        patterns = self.user_patterns[user_id]

        # Update patterns based on the provided interaction data.
        if 'response_length_preference' in interaction_data:
            patterns['preferred_response_length'] =
interaction_data['response_length_preference']

        if 'session_duration' in interaction_data:
            # Update typical session duration using a simple exponential moving average.
            current_avg = patterns['typical_session_duration']
            new_duration = interaction_data['session_duration']
            patterns['typical_session_duration'] = (current_avg * 0.8) + (new_duration * 0.2)
```

```
        if 'topics' in interaction_data and isinstance(interaction_data['topics'], list):
            # Track topic frequency (simple append for demonstration).
            for topic in interaction_data['topics']:
                if topic not in patterns['common_topics']:
                    patterns['common_topics'].append(topic)

        if 'complexity_preference' in interaction_data:
            patterns['complexity_preference'] = interaction_data['complexity_preference']

    def get_personalized_context(self, user_id: str) -> str:
        """
        Generates a personalized context string based on the tracked user interaction patterns.
        This context can then be injected into the LLM prompt.
        """
        if user_id not in self.user_patterns:
            return ""

        patterns = self.user_patterns[user_id]

        context = f"""
User Interaction Preferences for {user_id}:
- Prefers {patterns['preferred_response_length']} length responses.
- Typically engages for {patterns['typical_session_duration']:.1f} minutes.
- Common topics: {', '.join(patterns['common_topics'][:5])}.
- Complexity level: {patterns['complexity_preference']}.
"""
        return context.strip()

# Usage with pattern tracking
async def run_advanced_memory_system():
    advanced_memory = AdvancedMemorySystem()

    # Initialize ConversationChain with the combined memory.
    # Note: ConversationChain is a legacy component, but it directly demonstrates
CombinedMemory.
    # For modern LCEL, one would manually load memory variables and inject them into the
prompt.
    conversation = ConversationChain(
        llm=advanced_memory.llm,
        memory=advanced_memory.combined_memory,
        verbose=True
    )
```

```
    user_id = "sarah_123"

    # Track Sarah's initial interaction patterns.
    print(f"Tracking initial interaction patterns for {user_id}...")
    advanced_memory.track_interaction_pattern(user_id, {
        'response_length_preference': 'detailed',
        'session_duration': 15.5,
        'topics': ['machine learning', 'data science', 'python', 'neural networks'],
        'complexity_preference': 'advanced'
    })

    # Get personalized context based on tracked patterns.
    personal_context = advanced_memory.get_personalized_context(user_id)
    print(f"\nSarah's interaction style:\n{personal_context}")

    # Now, conversations can be tailored to Sarah's preferences by prepending the
personalized context.
    print("\nUser: How do I optimize my neural network?")
    response = await conversation.ainvoke( # Using ainvoke for asynchronous execution
        {"input": f"{personal_context}\n\nUser question: How do I optimize my neural network?"}
    )
    print(f"AI: {response['response']}")

    # Simulate another interaction with different patterns, updating Sarah's profile.
    print("\nTracking updated patterns for Sarah (shorter session, new topics)...")
    advanced_memory.track_interaction_pattern(user_id, {
        'session_duration': 5.0, # Shorter session
        'topics':
    })
    personal_context_updated = advanced_memory.get_personalized_context(user_id)
    print(f"\nSarah's updated interaction style:\n{personal_context_updated}")

    print("\nUser: What are best practices for deploying models to AWS?")
    response_updated = await conversation.ainvoke(
        {"input": f"{personal_context_updated}\n\nUser question: What are best practices for
deploying models to AWS?"}
    )
    print(f"AI: {response_updated['response']}")

# To run this example:
# asyncio.run(run_advanced_memory_system())
```

This system combines ConversationSummaryMemory, ConversationEntityMemory, and

ConversationKGMemory using LangChain's CombinedMemory.[17] While these LangChain components primarily handle textual conversation context and extracted entities, the track_interaction_pattern and get_personalized_context methods demonstrate how to integrate custom, non-textual (multi-modal) memory. This includes implicit user preferences and behavioral patterns, such as preferred response length, typical session duration, and common topics. By incorporating these implicit signals alongside explicit conversational content, the AI can tailor its responses not just based on what was said, but also *how* the user prefers to interact.

The development of "Multi-Modal Memory" represents a crucial step towards agents that are not merely conversational but truly intelligent and adaptive. By integrating implicit signals, such as a user's preferred response length or typical session duration, alongside explicit conversational content, the agent can develop a more holistic understanding of the user. This moves the agent's comprehension from simply "what was said" to a deeper understanding of "who is saying it and how they prefer to interact." This progression points to a future where AI agents evolve beyond being mere language models to become full-fledged "digital personas" capable of adapting to individual users. Such adaptation enables highly personalized and effective interactions across various modalities. Achieving this requires sophisticated data collection and integration beyond just chat logs, potentially involving user interface analytics, sentiment analysis, and comprehensive long-term user profiling.

4.2. Intelligent Memory Compression and Optimization

As AI agents accumulate vast amounts of information, efficiently managing this memory becomes critical for both performance and cost-effectiveness. Intelligent memory compression and optimization involve smart strategies to retain important information while discarding noise, often using techniques like importance scoring based on recency, frequency, user engagement, and keywords.

4.2.1. Memory Compression Logic

This custom IntelligentMemoryCompressor class implements a scoring mechanism to determine the importance of each memory entry. It then selectively retains a percentage of the most important memories and summarizes the less critical ones, actively managing the memory footprint.

Python

```
import json
from datetime import datetime, timedelta
from typing import Dict, List, Any
```

```
# import numpy as np # Often useful for more complex scoring/ranking, but not directly used
in this snippet

class IntelligentMemoryCompressor:
    """
    Manages and optimizes memory by intelligently compressing and summarizing less
important entries.
    """
    def __init__(self):
        self.compression_rules = {
            'importance_threshold': 0.6, # Memories below this score might be summarized or
discarded
            'recency_weight': 0.3, # Weight for how recent a memory is
            'frequency_weight': 0.4, # Weight for how frequently a topic/memory is mentioned
            'user_engagement_weight': 0.3, # Weight for user interaction with the memory
            'keyword_weight': 0.2 # Weight for presence of important keywords
        }
        # Internal store for memories. In a real system, this would be a database or vector store.
        self.memories_store: List] =

    def add_memory(self, memory_data: Dict[str, Any]):
        """
        Adds a new memory to the collection, ensuring necessary fields for scoring are present.
        """
        if 'timestamp' not in memory_data:
            memory_data['timestamp'] = datetime.now().isoformat()
        if 'mention_count' not in memory_data:
            memory_data['mention_count'] = 1
        self.memories_store.append(memory_data)

    def calculate_memory_importance(self, memory: Dict[str, Any]) -> float:
        """
        Calculates how important a memory is for retention based on defined rules.
        A higher score indicates greater importance.
        """
        # Recency score: Newer memories are generally more important. Decays over 30 days.
        memory_date = datetime.fromisoformat(memory['timestamp'])
        days_old = (datetime.now() - memory_date).days
        recency_score = max(0, 1 - (days_old / 30)) # Score from 1 (new) to 0 (30+ days old)

        # Frequency score: Memories mentioned more often are generally more important.
Capped at 1.0.
        frequency_score = min(1.0, memory.get('mention_count', 1) / 10) # Full score if mentioned
```

10+ times

```
        # User engagement score: Longer content or more user interaction implies higher
importance. Capped at 1.0.
        engagement_score = min(1.0, len(memory.get('content', '')) / 500) # Full score if content
is 500+ chars

        # Keyword score: Presence of specific important keywords increases importance.
        important_keywords = ['problem', 'project', 'deadline', 'important', 'urgent', 'remember',
'critical']
        keyword_score = sum(1 for keyword in important_keywords
                        if keyword in memory.get('content', '').lower()) / len(important_keywords) if
len(important_keywords) > 0 else 0.0

        # Weighted combination of all scores.
        rules = self.compression_rules
        total_score = (
            recency_score * rules['recency_weight'] +
            frequency_score * rules['frequency_weight'] +
            engagement_score * rules['user_engagement_weight'] +
            keyword_score * rules['keyword_weight']
        )

        return min(1.0, total_score) # Ensures score does not exceed 1.0

    def compress_memories(self, memories: List]) -> List]:
        """
        Intelligently compresses a list of memories by retaining important ones
        and summarizing less critical ones.
        """
        # Score all memories based on their calculated importance.
        scored_memories =
        for memory in memories:
            importance = self.calculate_memory_importance(memory)
            scored_memories.append((importance, memory))

        # Sort memories by importance in descending order.
        scored_memories.sort(key=lambda x: x, reverse=True)

        # Retain a percentage of the most important memories.
        # The article mentions "top 70% of important memories".
        keep_count = int(len(scored_memories) * 0.7)
        important_memories = [memory for _, memory in scored_memories[:keep_count]]
```

```
        # Summarize the remaining less important memories.
        less_important = [memory for _, memory in scored_memories[keep_count:]]
        if less_important:
            summary_content = self.create_memory_summary(less_important)
            important_memories.append({
                'type': 'summary', # Flagging this entry as a summary
                'content': summary_content,
                'timestamp': datetime.now().isoformat(),
                'original_count': len(less_important),
                'is_compressed_summary': True # Custom flag for easy identification
            })

        return important_memories

    def create_memory_summary(self, memories: List]) -> str:
        """
        Creates a simplified summary of multiple memories.
        In a production system, this would typically involve an LLM for sophisticated
summarization.
        """
        topics = {}
        for memory in memories:
            content = memory.get('content', '')
            # Simple topic extraction: count words longer than 4 characters and are alphabetic.
            words = content.lower().split()
            for word in words:
                if len(word) > 4 and word.isalpha():
                    topics[word] = topics.get(word, 0) + 1

        # Get the top 5 most common topics.
        common_topics = sorted(topics.items(), key=lambda item: item, reverse=True)[:5] # Sort
by count
        topic_list = [topic for topic, count in common_topics]

        if not topic_list:
            return f"Summary of {len(memories)} less important conversations."

        return f"Summary of {len(memories)} conversations covering: {', '.join(topic_list)}."

# Usage example
async def run_memory_compression_demo():
    compressor = IntelligentMemoryCompressor()
```

```
    # Simulate a large collection of memories with varying importance.
    large_memory_collection =

    # Add simulated memories to the compressor's internal store.
    for mem in large_memory_collection:
        compressor.add_memory(mem)

    print(f"Original memory count: {len(compressor.memories_store)} individual memories.")

    # Compress the memories intelligently.
    optimized_memories = compressor.compress_memories(compressor.memories_store)
    print(f"Compressed {len(compressor.memories_store)} memories into
{len(optimized_memories)} entries (retained {int(len(compressor.memories_store) * 0.7)}
important ones and generated a summary).")

    print("\n--- Optimized Memories (Content Snippets) ---")
    for i, mem in enumerate(optimized_memories):
        content_preview = mem['content'][:80] + '...' if len(mem['content']) > 80 else
mem['content']
        importance_str = f", Importance: {compressor.calculate_memory_importance(mem):.2f}"
if 'type' not in mem else ""
        print(f"Entry {i+1}: Type: {mem.get('type', 'detail')}{importance_str}, Content:
'{content_preview}'")

# To run this example:
# asyncio.run(run_memory_compression_demo())
```

This custom IntelligentMemoryCompressor class implements a sophisticated scoring mechanism to evaluate the importance of each memory entry. The score is derived from a weighted combination of factors including recency, frequency of mention, user engagement (inferred from content length), and the presence of critical keywords. Based on these scores, the system intelligently retains a specified percentage of the most important memories in their original detail and then generates a concise summary of the less important ones. This active management of the memory footprint directly contributes to cost savings by reducing the volume of data that needs to be stored and processed, while simultaneously improving the performance of memory retrieval by focusing on salient information.

The concept of "memory compression and optimization" signifies that memory in AI agents is not a static repository but an actively managed resource. The IntelligentMemoryCompressor dynamically evaluates and transforms memories, deciding whether to *retain* them in full, *summarize* them, or implicitly *discard* them based on their perceived value. This represents a lifecycle management approach, similar to data lifecycle management in traditional systems, but applied directly to the AI agent's knowledge base. This progression suggests that advanced AI memory systems will increasingly incorporate intelligent, LLM-driven (or hybrid) data lifecycle management. This moves beyond simple Time-To-Live (TTL) or fixed-window policies to a more nuanced, value-based approach, ensuring that the most salient information is retained and efficiently accessible, while less critical data is compressed or archived. This directly impacts both the operational cost and the retrieval performance of the AI agent.

4.3. Team Memory: Collaborative AI Agents

The concept of "Team Memory" represents a significant leap in AI agent capabilities, enabling multiple agents to share knowledge, learn from each other's interactions, and collaborate to provide more informed and personalized user experiences. This architecture often involves shared memory stores and sophisticated mechanisms for conflict resolution.

4.3.1. Agno's Collaborative Memory Architecture

Agno is particularly well-suited for building multi-agent systems with collaborative memory, offering built-in SharedMemory and Crew components for orchestration and real-time synchronization.[1]

Python

```python
from agno import Agent, Crew # Crew for multi-agent orchestration
from agno.memory import VectorMemory, SharedMemory # Agno's memory components
from agno.models.openai import OpenAIChat # Agno's OpenAI model wrapper [13]
import asyncio
from typing import Dict, List, Any
from datetime import datetime
import json

class CollaborativeMemorySystem:
  """
  An advanced collaborative memory system using Agno for multi-agent teams.
  Agents share knowledge through a central shared memory.
  """
  def __init__(self, crew_name: str):
    self.crew_name = crew_name

    # Initialize a SharedMemory instance for the entire crew.
    # This is the central repository where agents store and retrieve collective knowledge.
```

```
    self.shared_memory = SharedMemory(
      collection_name=f"crew_{crew_name}_shared_knowledge",
      embeddings_model="text-embedding-3-large", # High-quality embedding model
      sync_strategy="real_time" # Configures memories to synchronize instantly
    )

    # Dictionaries to hold individual agent memories and agent instances.
    self.agent_memories: Dict[str, VectorMemory] = {}
    self.agents: Dict[str, Agent] = {}

    # Log for tracking collaboration patterns for analytics.
    self.collaboration_log: List] =

  async def add_agent(self, agent_id: str, role: str, expertise: List[str]) -> Agent:
    """
    Adds an agent to the collaborative system, configuring its personal memory
    to synchronize with the shared team memory.
    """
    # Create agent-specific memory that syncs with shared memory.
    # By setting `parent_memory`, individual agent memories contribute to the shared pool.
    agent_memory = VectorMemory(
      collection_name=f"agent_{agent_id}_personal_memory",
      embeddings_model="text-embedding-3-large",
      parent_memory=self.shared_memory # Establishes synchronization with team memory
    )

    # Create the collaborative agent using Agno's Agent class.
    agent = Agent(
      model=OpenAIChat(id="gpt-4o", temperature=0.7), # Updated model to gpt-4o [13]
      memory=agent_memory, # Assign the individual agent's memory
      instructions=f"""
      You are a {role} working as part of the {self.crew_name} team.

...[source](https://medium.com/%40nomannayeem/building-ai-agents-that-actually-rememb
er-a-developers-guide-to-memory-management-in-2025-062fd0be80a1) teammates by
adding it to memory.
      """
    )

    self.agents[agent_id] = agent
    self.agent_memories[agent_id] = agent_memory
    return agent
```

```
  async def collaborative_response(self, user_id: str, query: str,
                                   primary_agent_id: str) -> Dict[str, Any]:
    """
    Orchestrates a response generation with full team collaboration.
    """
    if primary_agent_id not in self.agents:
      raise ValueError(f"Primary agent '{primary_agent_id}' not found.")

    primary_agent = self.agents[primary_agent_id]

    # Step 1: Retrieve relevant context from the shared team knowledge base.
    team_context = await self._get_team_context(user_id, query)

    # Step 2: Identify which other agents should contribute based on the query and their
expertise.
    contributing_agents_ids = await self._identify_contributing_agents(query,
primary_agent_id)

    # Step 3: Gather input from identified contributing agents.
    agent_inputs = {}
    for agent_id in contributing_agents_ids:
      if agent_id!= primary_agent_id: # The primary agent handles the main response.
        agent_input = await self._get_agent_input(agent_id, query, team_context)
        agent_inputs[agent_id] = agent_input

    # Step 4: The primary agent generates the final response, incorporating team input.
    collaboration_context = self._build_collaboration_context(agent_inputs, team_context)

    full_prompt_for_primary_agent = f"""
    You are the primary agent ({primary_agent_id}) responsible for responding to the user.
    Team Context: {collaboration_context}

    User Query: {query}

    Provide a comprehensive response that incorporates team knowledge and expertise.
    """
    response_result = await primary_agent.run(full_prompt_for_primary_agent)
    final_response = response_result.output

    # Step 5: Share learnings from this interaction with the team's shared memory.
    await self._share_interaction_learnings(user_id, query, final_response, primary_agent_id)

    # Step 6: Log the collaboration for analytics and performance monitoring.
```

```
        self._log_collaboration(user_id, primary_agent_id, contributing_agents_ids, query)

        return {
            'response': final_response,
            'primary_agent': primary_agent_id,
            'contributing_agents': contributing_agents_ids,
            'team_context_used': len(team_context) > 0,
            'collaboration_score': len(contributing_agents_ids) / len(self.agents) if self.agents else
0
        }

    async def _get_team_context(self, user_id: str, query: str) -> str:
        """
        Retrieves relevant context from the shared team memory based on user and query.
        """
        # Search shared memory for information relevant to the user and the current query.
        relevant_memories = await self.shared_memory.search(
            query=f"user:{user_id} {query}",
            limit=10, # Retrieve top 10 relevant memories
            include_metadata=True # Include metadata like agent_id and timestamp
        )

        if not relevant_memories:
            return "No previous team interactions with this user."

        context_parts =
        context_parts.append(f"Previous team interactions with user {user_id}:")

        # Build context from the most relevant team memories.
        for memory in relevant_memories[:5]: # Limit context to top 5 for brevity in the prompt.
            agent_id = memory.get('metadata', {}).get('agent_id', 'unknown')
            timestamp = memory.get('metadata', {}).get('timestamp', 'unknown')
            content = memory.get('content', '')[:150] # Truncate content for brevity.

            context_parts.append(f" - {agent_id} ({timestamp}): {content}...")

        return "\n".join(context_parts)

    async def _identify_contributing_agents(self, query: str, primary_agent_id: str) -> List[str]:
        """
        Identifies which agents should contribute to the response based on the query and their
expertise.
        """
```

```python
    contributing_agents = [primary_agent_id] # Primary agent always contributes.

    query_lower = query.lower()

    for agent_id, agent in self.agents.items():
        if agent_id == primary_agent_id:
            continue # Skip primary agent for identifying *additional* contributors.

        # Extract agent's expertise from instructions (simplified parsing).
        agent_instructions = agent.instructions or ""
        # Define example expertise keywords for classification.
        expertise_keywords = {
            'technical': ['bug', 'error', 'technical', 'code', 'api', 'performance'],
            'sales': ['price', 'upgrade', 'purchase', 'plan', 'billing', 'enterprise'],
            'support': ['help', 'problem', 'issue', 'trouble', 'support'],
            'product': ['feature', 'functionality', 'product', 'roadmap', 'analytics']
        }

        for expertise_type, keywords in expertise_keywords.items():
            if expertise_type in agent_instructions.lower():
                if any(keyword in query_lower for keyword in keywords):
                    contributing_agents.append(agent_id)
                    break # Found relevant expertise, move to next agent.

    return list(set(contributing_agents)) # Return unique list of contributing agents.

async def _get_agent_input(self, agent_id: str, query: str, team_context: str) -> str:
    """
    Retrieves expert input or advice from a contributing agent.
    """
    agent = self.agents[agent_id]

    # Prompt the contributing agent for their perspective.
    input_response = await agent.run(
        f"""
        Team Context: {team_context}

        User Query: {query}

        As a team member, provide your expert perspective or advice for handling this query.
        Keep your input concise and focused on your area of expertise.
        If this query is outside your expertise, state that briefly.
        """
```

```
        )

        return input_response.output

    def _build_collaboration_context(self, agent_inputs: Dict[str, str], team_context: str) -> str:
        """
        Constructs the collaboration context string from individual agent inputs and shared team
context.
        """
        context_parts =
        if team_context:
            context_parts.append(team_context)

        if agent_inputs:
            context_parts.append("\nInput from teammates:")
            for agent_id, input_text in agent_inputs.items():
                agent_role_line = next((line for line in self.agents[agent_id].instructions.split('\n') if
"You are a" in line), "Team Member")
                agent_role = agent_role_line.split('You are a ').split(' working') if 'You are a' in
agent_role_line else "Team Member"
                context_parts.append(f"- {agent_role} ({agent_id}): {input_text}")
        else:
            context_parts.append("No additional team input for this query.")

        return "\n".join(context_parts)

    async def _share_interaction_learnings(self, user_id: str, query: str, response: str, agent_id:
str):
        """
        Shares what was learned from the interaction with the team's shared memory.
        """
        # Create a memory entry for team knowledge.
        memory_entry_content = f"User Query: {query}\nResponse: {response[:200]}..." #
Truncate response for memory.
        memory_metadata = {
            'user_id': user_id,
            'agent_id': agent_id,
            'timestamp': datetime.now().isoformat(),
            'interaction_type': 'collaborative_response'
        }
        # Store in shared memory. Agno's add_memory handles embedding and storage.
        await self.shared_memory.add_memory(content=memory_entry_content,
metadata=memory_metadata)
```

```
    def _log_collaboration(self, user_id: str, primary_agent: str, contributors: List[str], query: str):
        """
        Logs collaboration patterns for analysis and auditing.
        """
        collaboration_entry = {
            'timestamp': datetime.now().isoformat(),
            'user_id': user_id,
            'primary_agent': primary_agent,
            'contributing_agents': contributors,
            'collaboration_level': len(contributors),
            'query_type': self._classify_query(query)
        }
        self.collaboration_log.append(collaboration_entry)

    def _classify_query(self, query: str) -> str:
        """Simple query classification for logging purposes."""
        query_lower = query.lower()
        if any(word in query_lower for word in ['bug', 'error', 'broken', 'issue', 'performance']):
            return 'technical_issue'
        elif any(word in query_lower for word in ['price', 'cost', 'upgrade', 'billing', 'plan']):
            return 'sales_inquiry'
        elif any(word in query_lower for word in ['how', 'tutorial', 'guide', 'explain']):
            return 'educational'
        else:
            return 'general_inquiry'

    async def get_collaboration_analytics(self) -> Dict[str, Any]:
        """
        Generates insights about team collaboration patterns from the log.
        """
        if not self.collaboration_log:
            return {"message": "No collaborations recorded yet"}

        total_collaborations = len(self.collaboration_log)
        avg_collaboration_level = sum(entry['collaboration_level'] for entry in
self.collaboration_log) / total_collaborations

        # Count collaborations per agent.
        agent_collaboration_count = {}
        for entry in self.collaboration_log:
            agent_id = entry['primary_agent']
            agent_collaboration_count[agent_id] = agent_collaboration_count.get(agent_id, 0) + 1
```

```
        # Distribution of query types.
        query_types = {}
        for entry in self.collaboration_log:
            query_type = entry['query_type']
            query_types[query_type] = query_types.get(query_type, 0) + 1

        return {
            'total_collaborations': total_collaborations,
            'average_agents_per_collaboration': avg_collaboration_level,
            'most_collaborative_agent': max(agent_collaboration_count,
key=agent_collaboration_count.get) if agent_collaboration_count else None,
            'query_type_distribution': query_types,
            'team_efficiency': avg_collaboration_level / len(self.agents) if self.agents else 0 # How
well the team collaborates relative to its size
        }

# Usage example: Building a customer success team
async def collaborative_team_demo():
    # Create collaborative memory system.
    team_memory = CollaborativeMemorySystem("customer_success_team")

    # Add agents with different expertise.
    support_agent = await team_memory.add_agent( # Await add_agent as it creates internal
memory
        "support_specialist",
        "Technical Support Specialist",
        ["technical_issues", "troubleshooting", "bug_reports", "API performance"]
    )

    sales_agent = await team_memory.add_agent(
        "sales_rep",
        "Sales Representative",
        ["pricing", "upgrades", "product_demos", "billing", "enterprise plans"]
    )

    product_agent = await team_memory.add_agent(
        "product_manager",
        "Product Manager",
        ["feature_requests", "roadmap", "product_feedback", "analytics platform"]
    )

    print("=== Collaborative Team Demonstration ===")
```

```
    # Simulate a technical query that might need multiple perspectives (support and sales).
    result1 = await team_memory.collaborative_response(
        user_id="enterprise_client_001",
        query="We're experiencing performance issues with the API and considering upgrading
our plan. What are our options?",
        primary_agent_id="support_specialist"
    )

    print(f"\n--- Interaction 1 ---")
    print(f"Primary Agent: {result1['primary_agent']}")
    print(f"Contributing Agents: {result1['contributing_agents']}")
    print(f"Response: {result1['response'][:300]}...")
    print(f"Collaboration Score: {result1['collaboration_score']:.2f}")

    print("\n" + "="*50 + "\n")

    # Simulate a sales query with potential technical considerations (sales and product).
    result2 = await team_memory.collaborative_response(
        user_id="enterprise_client_001",
        query="What's included in the enterprise plan and will it solve our API performance
issues?",
        primary_agent_id="sales_rep"
    )

    print(f"\n--- Interaction 2 ---")
    print(f"Primary Agent: {result2['primary_agent']}")
    print(f"Contributing Agents: {result2['contributing_agents']}")
    print(f"Response: {result2['response'][:300]}...")
    print(f"Collaboration Score: {result2['collaboration_score']:.2f}")

    # Get team analytics.
    analytics = await team_memory.get_collaboration_analytics()
    print(f"\n=== Team Collaboration Analytics ===")
    print(f"Total Collaborations: {analytics['total_collaborations']}")
    print(f"Average Agents per Collaboration:
{analytics['average_agents_per_collaboration']:.1f}")
    print(f"Most Collaborative Agent (Primary): {analytics['most_collaborative_agent']}")
    print(f"Query Type Distribution: {analytics['query_type_distribution']}")
    print(f"Team Efficiency (Avg Agents / Total Agents): {analytics['team_efficiency']:.2f}")

# To run this example:
# asyncio.run(collaborative_team_demo())
```

Agno's CollaborativeMemorySystem facilitates multi-agent collaboration through a SharedMemory instance, which acts as a central knowledge repository for the entire team.[1] Each individual agent is configured with its own
VectorMemory that automatically synchronizes with this shared pool by setting parent_memory=self.shared_memory. The collaborative_response method orchestrates the entire process: it retrieves relevant context from the shared team memory, dynamically identifies which agents should contribute based on their expertise and the user's query, gathers input from these contributing agents, and then has the primary agent synthesize a comprehensive response. This ensures that agents benefit from collective intelligence, leading to more informed and personalized user interactions. Agno's design, particularly with its Crew class, aims to simplify the orchestration of complex multi-agent workflows, making it a robust choice for production-grade collaborative AI systems.[5]
The architectural design of Agno, with its direct integration of VectorMemory into the Agent class and the provision of SharedMemory for team collaboration, represents a higher level of abstraction for building agents.[1] This design choice is particularly beneficial for teams, as it provides opinionated, built-in solutions for common needs like vector memory and multi-agent synchronization. This allows developers to focus more on the specific logic and roles of their agents rather than on the underlying infrastructure, significantly accelerating the development and deployment of sophisticated AI applications. For projects requiring robust, out-of-the-box features for complex multi-agent systems, frameworks like Agno offer a streamlined experience by abstracting away much of the underlying plumbing.

4.3.2. Memory Conflict Resolution

When multiple AI agents share and update a common memory, conflicts can arise if they store contradictory information about the same fact or user preference. Effective conflict resolution strategies are essential to maintain the integrity and consistency of the shared knowledge base.

Python

```python
from typing import Dict, List, Tuple, Optional
from datetime import datetime
import json
import asyncio # For async methods

class MemoryConflictResolver:
    """
    Handles conflicts that may arise when multiple agents have different information about
```

users or facts.
```
    """
    def __init__(self):
        self.resolution_strategies = {
            'recency': self._resolve_by_recency,
            'authority': self._resolve_by_authority,
            'consensus': self._resolve_by_consensus,
            'confidence': self._resolve_by_confidence
        }

    async def detect_conflicts(self, memories: List]) -> List]:
        """
        Detects conflicts within a list of memory entries.
        Conflicts are identified by grouping memories by user and a defined 'fact_type',
        then checking for differing 'value' fields within each group.
        """
        conflicts =

        # Group memories by a composite key (user_id and fact_type) to find potential conflicts.
        memory_groups = {}
        for memory in memories:
            user_id = memory.get('user_id')
            fact_type = memory.get('fact_type', 'general') # Default to 'general' if not specified

            key = f"{user_id}:{fact_type}"
            if key not in memory_groups:
                memory_groups[key] =
            memory_groups[key].append(memory)

        # Iterate through groups and identify actual conflicts.
        for group_key, group_memories in memory_groups.items():
            if len(group_memories) > 1: # A conflict can only exist if there's more than one memory
for a fact.
                conflict_details = self._check_for_conflict(group_memories)
                if conflict_details:
                    conflicts.append({
                        'group_key': group_key,
                        'conflicting_memories': group_memories,
                        'conflict_type': conflict_details['type'],
                        'severity': conflict_details['severity']
                    })

        return conflicts
```

```
    def _check_for_conflict(self, memories: List]) -> Optional]:
        """
        Checks if a group of memories contains conflicting information, specifically by comparing
'value' fields.
        """
        values =
        for memory in memories:
            if 'value' in memory:
                values.append(memory['value'])

        # A conflict exists if there are multiple unique values for the same fact type.
        unique_values = set(values)
        if len(unique_values) > 1 and len(values) > 1:
            return {
                'type': 'value_mismatch',
                'severity': 'medium' if len(unique_values) <= 3 else 'high' # Severity based on number
of differing values
            }

        return None

    async def resolve_conflict(self, conflict: Dict[str, Any], strategy: str = 'recency') -> Dict[str,
Any]:
        """
        Resolves a memory conflict using a specified strategy.
        Defaults to 'recency' if an unknown strategy is provided.
        """
        if strategy not in self.resolution_strategies:
            print(f"Warning: Unknown conflict resolution strategy '{strategy}'. Falling back to
'recency'.")
            strategy = 'recency' # Default fallback strategy

        resolution_func = self.resolution_strategies[strategy]
        resolved_memory = await resolution_func(conflict['conflicting_memories'])

        return {
            'resolved_memory': resolved_memory,
            'strategy_used': strategy,
            'conflicting_count': len(conflict['conflicting_memories']),
            'resolution_timestamp': datetime.now().isoformat()
        }
```

```
    async def _resolve_by_recency(self, memories: List]) -> Dict[str, Any]:
        """Resolves conflict by selecting the most recent memory based on its timestamp."""
        # Sort memories by timestamp in descending order (most recent first).
        sorted_memories = sorted(
            memories,
            key=lambda x: x.get('timestamp', '1970-01-01T00:00:00Z'), # Default to old date if no
timestamp
            reverse=True
        )

        most_recent = sorted_memories # The first element is the most recent.
        most_recent['resolution_method'] = 'recency'
        most_recent['confidence_score'] = 0.8 # Assign a high confidence to recent information.

        return most_recent

    async def _resolve_by_authority(self, memories: List]) -> Dict[str, Any]:
        """
        Resolves conflict by selecting the memory from the agent with the highest defined
authority level.
        """
        # Define an example agent authority hierarchy (customize based on application's needs).
        authority_levels = {
            'admin': 10,
            'product_manager': 8,
            'support_specialist': 7,
            'sales_rep': 6,
            'general_agent': 5,
            'user_input': 1 # User input might be lowest authority for factual conflicts
        }

        highest_authority_memory = None
        highest_level = -1

        for memory in memories:
            agent_role = memory.get('agent_role', 'general_agent') # Get the role of the agent that
created the memory.
            authority_level = authority_levels.get(agent_role, 1) # Default to low authority if role is
unknown.

            if authority_level > highest_level:
                highest_level = authority_level
                highest_authority_memory = memory
```

```
        if highest_authority_memory:
            highest_authority_memory['resolution_method'] = 'authority'
            highest_authority_memory['confidence_score'] = 0.9 # Assign high confidence to
authoritative information.
            return highest_authority_memory

        # Fallback to recency if no authority information is available or applicable.
        return await self._resolve_by_recency(memories)

    async def _resolve_by_consensus(self, memories: List]) -> Dict[str, Any]:
        """
        Resolves conflict by identifying the most common value among conflicting memories
(consensus).
        """
        value_counts = {}
        for memory in memories:
            value = memory.get('value')
            if value is not None:
                value_counts[value] = value_counts.get(value, 0) + 1

        if value_counts:
            # Find the value that appears most frequently.
            consensus_value = max(value_counts, key=value_counts.get)
            consensus_count = value_counts[consensus_value]

            # Return one of the memories that holds the consensus value.
            for memory in memories:
                if memory.get('value') == consensus_value:
                    memory['resolution_method'] = 'consensus'
                    # Confidence score reflects the proportion of memories supporting the
consensus.
                    memory['confidence_score'] = min(0.95, consensus_count / len(memories))
                    memory['consensus_support'] = consensus_count
                    return memory

        # Fallback if no clear consensus is found (e.g., all values are unique, or no 'value' field).
        return await self._resolve_by_recency(memories)

    async def _resolve_by_confidence(self, memories: List]) -> Dict[str, Any]:
        """
        Resolves conflict by selecting the memory with the highest explicitly provided confidence
score.
```

"""
# Sort memories by their 'confidence_score' in descending order.
sorted_memories = sorted(
    memories,
    key=lambda x: x.get('confidence_score', 0.5), # Default confidence if not provided
    reverse=True
)

highest_confidence = sorted_memories # The first element is the one with highest confidence.
highest_confidence['resolution_method'] = 'confidence'
Its confidence score is already inherent or set by the agent that created it.

return highest_confidence

This MemoryConflictResolver class provides a framework for detecting and resolving inconsistencies in shared memory. It identifies conflicts by grouping memories by user and fact type, then checking for differing values within those groups. The class supports multiple resolution strategies:

- **Recency:** Prioritizes the most recently updated memory.
- **Authority:** Selects the memory contributed by an agent with a higher predefined authority level.
- **Consensus:** Chooses the value that appears most frequently among the conflicting memories.
- **Confidence:** Selects the memory that has the highest associated confidence score.

These strategies are crucial for maintaining a consistent and reliable shared knowledge base in multi-agent systems.

4.3.3. Integration with Team Memory System (Conflict Resolution)

The MemoryConflictResolver can be integrated into a team memory system to automatically detect and resolve conflicts as memories are updated.

Python

from typing import Dict, Any, List
from datetime import datetime
import asyncio # For async operations
# Assume SharedMemoryStore is a simplified representation of Agno's SharedMemory or a similar concept.
# For this example, we'll create a basic mock SharedMemoryStore.

```
class SharedMemoryStore:
    """
    A simplified mock of a shared memory store for demonstration purposes.
    In a real system, this would be backed by a persistent database.
    """
    def __init__(self):
        self.user_memories: Dict] = {} # Stores memories organized by user_id

    def update_user_memory(self, user_id: str, agent_id: str, memory_data: Dict[str, Any]):
        """
        Updates a user's memory in the store.
        This is a basic add/update, without conflict resolution.
        """
        if user_id not in self.user_memories:
            self.user_memories[user_id] = {'interactions':}

        # Add metadata for conflict resolution (e.g., agent_id, timestamp, confidence_score,
agent_role)
        memory_data['agent_id'] = agent_id
        if 'timestamp' not in memory_data:
            memory_data['timestamp'] = datetime.now().isoformat()
        # Assume agent_role is passed or derived from agent_id for authority resolution
        if 'agent_role' not in memory_data:
            memory_data['agent_role'] = 'general_agent' # Default role

        self.user_memories[user_id]['interactions'].append(memory_data)
        print(f"Memory added for user {user_id} by {agent_id}: {memory_data.get('content',
memory_data.get('value', ''))}")

    def get_user_memories(self, user_id: str) -> List]:
        """Retrieves all memories for a specific user."""
        return self.user_memories.get(user_id, {}).get('interactions',)

    def replace_user_memory(self, user_id: str, old_memory: Dict[str, Any], new_memory:
Dict[str, Any]):
        """
        Replaces an old memory with a new, resolved memory.
        In a real system, this would involve updating a database record.
        For this mock, it's a simplified replacement.
        """
        if user_id in self.user_memories:
            interactions = self.user_memories[user_id]['interactions']
```

```
        try:
          # Find and remove the old memory, then add the new one.
          # This is a simplified approach; in production, you might identify by unique ID.
          interactions.remove(old_memory)
          interactions.append(new_memory)
          print(f"Memory for user {user_id} resolved and updated.")
        except ValueError:
          print(f"Warning: Old memory not found for replacement for user {user_id}.")

# Re-importing MemoryConflictResolver for direct use in this context.
# from.memory_conflict_resolver import MemoryConflictResolver # Assuming it's in a
separate file
# For this example, we will define it inline or ensure it's available.
# (The MemoryConflictResolver class from 4.3.2 is assumed to be defined and available.)

class ConflictAwareTeamMemory(SharedMemoryStore):
  """
  An enhanced team memory system that integrates conflict detection and resolution.
  """
  def __init__(self):
    super().__init__()
    self.conflict_resolver = MemoryConflictResolver()
    self.conflict_log: List] = # Log of detected and resolved conflicts

  async def update_user_memory_safe(self, user_id: str, agent_id: str, memory_data: Dict[str,
Any]):
    """
    Updates memory, performing conflict detection and resolution if necessary.
    """
    # First, add the new memory normally (this might create a conflict).
    self.update_user_memory(user_id, agent_id, memory_data)

    # Retrieve all current memories for the user to check for conflicts.
    user_memories = self.get_user_memories(user_id)
    conflicts = await self.conflict_resolver.detect_conflicts(user_memories)

    # Resolve any conflicts found.
    for conflict in conflicts:
      print(f"Conflict detected for {conflict['group_key']}: {conflict['conflict_type']}")
      # Resolve using a chosen strategy (e.g., 'recency', 'authority', 'consensus', 'confidence')
      resolution = await self.conflict_resolver.resolve_conflict(conflict, strategy='confidence')
# Example using confidence
      resolved_memory = resolution['resolved_memory']
```

```
        # Log the conflict and its resolution.
        self.conflict_log.append({
            'user_id': user_id,
            'conflict': conflict,
            'resolution': resolution,
            'timestamp': datetime.now().isoformat()
        })

        # Update the memory store with the resolved information.
        # This requires removing the conflicting memories and adding the resolved one.
        # For simplicity in this mock, we'll just replace the conflicting ones with the resolved
one.
        # In a real system, you'd have unique IDs for memories and update/delete specific
records.
        # Here, we'll assume the resolved memory is the 'correct' one and remove all others for
that fact_type.

        # Simplified application of resolution: remove all conflicting memories of this fact_type
        # and add the resolved one.
        fact_type_to_clear = conflict['group_key'].split(':')
        self.user_memories[user_id]['interactions'] = [
            m for m in self.user_memories[user_id]['interactions']
            if not (m.get('fact_type') == fact_type_to_clear and m.get('user_id') == user_id)
        ]
        self.user_memories[user_id]['interactions'].append(resolved_memory)
        print(f"Conflict resolved for {user_id} using '{resolution['strategy_used']}' strategy.
Resolved value: {resolved_memory.get('value', resolved_memory.get('content', ''))[:50]}...")


    def get_conflict_statistics(self) -> Dict[str, Any]:
        """
        Provides statistics about memory conflicts detected and resolved.
        """
        if not self.conflict_log:
            return {"conflicts_detected": 0, "message": "No conflicts detected or logged yet."}

        total_conflicts = len(self.conflict_log)

        # Count usage of each resolution strategy.
        strategy_usage = {}
        for log_entry in self.conflict_log:
            strategy = log_entry['resolution']['strategy_used']
```

```
            strategy_usage[strategy] = strategy_usage.get(strategy, 0) + 1

        # Calculate average conflicts per user, if user memories exist.
        avg_conflicts_per_user = total_conflicts / len(self.user_memories) if self.user_memories
else 0

        return {
            'conflicts_detected': total_conflicts,
            'resolution_strategies_used': strategy_usage,
            'average_conflicts_per_user': avg_conflicts_per_user,
            'recent_conflicts': self.conflict_log[-5:] # Show last 5 conflicts for quick review
        }

# Usage example
async def run_conflict_aware_team_memory():
    conflict_aware_memory = ConflictAwareTeamMemory()

    # Simulate conflicting information from different agents.
    print("--- Simulating Conflicting Memory Updates ---")
    await conflict_aware_memory.update_user_memory_safe(
        "user123",
        "support_agent",
        {"fact_type": "user_preference", "value": "detailed_responses", "confidence_score": 0.7,
"agent_role": "support_specialist"}
    )
    await asyncio.sleep(0.1) # Simulate slight delay

    await conflict_aware_memory.update_user_memory_safe(
        "user123",
        "sales_agent",
        {"fact_type": "user_preference", "value": "brief_responses", "confidence_score": 0.8,
"agent_role": "sales_rep"}
    )
    await asyncio.sleep(0.1)

    await conflict_aware_memory.update_user_memory_safe(
        "user123",
        "admin_agent",
        {"fact_type": "user_preference", "value": "detailed_responses", "confidence_score": 0.95,
"agent_role": "admin"} # Admin has highest authority
    )
    await asyncio.sleep(0.1)
```

```
    # Check the final state of user123's memories for 'user_preference'
    print("\n--- Final State of User123's 'user_preference' Memory ---")
    final_memories = conflict_aware_memory.get_user_memories("user123")
    for mem in final_memories:
        if mem.get('fact_type') == 'user_preference':
            print(f"  - Value: {mem.get('value')}, Source: {mem.get('agent_id')}, Resolved by:
{mem.get('resolution_method', 'N/A')}")

    # Check conflict statistics.
    stats = conflict_aware_memory.get_conflict_statistics()
    print(f"\n--- Conflict Statistics ---")
    print(json.dumps(stats, indent=2))

# To run this example:
# asyncio.run(run_conflict_aware_team_memory())
```

This ConflictAwareTeamMemory class extends a basic SharedMemoryStore by integrating the MemoryConflictResolver. When an agent updates a user's memory via update_user_memory_safe, the system automatically checks for conflicts among existing memories for that user and fact type. If conflicts are detected, the MemoryConflictResolver is invoked with a specified strategy (e.g., confidence), and the memory store is updated with the resolved information. This process is logged, providing an audit trail and statistics on conflict resolution. This robust mechanism is vital for maintaining data integrity and ensuring consistent agent behavior in collaborative AI environments.

4.4. Taking It to Production: Essential Considerations

Deploying memory-enabled AI systems at scale requires addressing several critical production challenges beyond core memory implementation. These include scalable storage, comprehensive monitoring, cost optimization, and robust security and privacy practices.

4.4.1. Scalable Memory Storage

Production-grade memory systems demand scalable and reliable storage solutions. This often involves a multi-tiered approach, combining high-speed caching layers (like Redis) for frequently accessed data with robust persistent databases (like PostgreSQL) for long-term storage.

Python

```
import asyncio
import redis.asyncio as redis # Using redis.asyncio for async operations [18, 19, 20]
import sqlite3 # Using SQLite for simplified DB demo, but PostgreSQL is mentioned for prod
import json
from typing import Dict, List, Any, Optional
from datetime import datetime, timedelta
import logging
from dataclasses import dataclass
from concurrent.futures import ThreadPoolExecutor
import hashlib # For hashing user IDs and cache keys
import os # For SQLite file management

@dataclass
class MemoryConfig:
    """Configuration for a production memory system."""
    redis_url: str = "redis://localhost:6379"
    # In production, this would be a proper database connection string like PostgreSQL
    db_file: str = "production_memory.db" # Using SQLite for local demo
    max_memory_per_user: int = 10000 # Maximum number of memories to store per user
    memory_ttl_days: int = 90 # Time-to-live for memories before potential cleanup/archival
    batch_size: int = 100 # Size for batch operations to improve efficiency
    max_concurrent_operations: int = 50 # Max workers for thread pool executor
    enable_encryption: bool = True # Flag to enable/disable data encryption
    backup_interval_hours: int = 6 # Frequency for data backups (conceptual)

class ProductionMemoryStore:
    """
    A production-ready memory storage system incorporating caching, persistence,
    validation, and basic security features.
    """
    def __init__(self, config: MemoryConfig):
        self.config = config
        self.redis_client: Optional = None
        self.db_connection: Optional[sqlite3.Connection] = None # Using SQLite for simplicity
        self.executor = ThreadPoolExecutor(max_workers=config.max_concurrent_operations)

        # Monitoring and metrics for operations.
        self.operation_counts = {
            'reads': 0,
            'writes': 0,
            'errors': 0,
            'cache_hits': 0,
```

```
            'cache_misses': 0
        }

        # Setup logging for production insights.
        logging.basicConfig(level=logging.INFO)
        self.logger = logging.getLogger('ProductionMemory')

    async def initialize(self):
        """Initializes all connections and resources for the memory store."""
        try:
            # Initialize Redis for caching.
            try:
                self.redis_client = redis.from_url(
                    self.config.redis_url,
                    decode_responses=True,
                    max_connections=20, # Connection pool size
                    retry_on_timeout=True
                )
                await self.redis_client.ping() # Test Redis connection
                self.logger.info("Redis cache connected successfully.")
            except redis.ConnectionError as e:
                self.logger.warning(f"Redis connection failed ({e}). Running without cache.")
                self.redis_client = None # Disable Redis if connection fails

            # Initialize database (using SQLite for local demo).
            await self._initialize_database()

            self.logger.info("Production memory store initialized successfully.")

        except Exception as e:
            self.logger.error(f"Failed to initialize memory store: {e}")
            raise # Re-raise to indicate critical failure

    async def _initialize_database(self):
        """Initializes the database schema."""
        # In a production environment, this would use a proper database connection pool
        # and a more robust schema for PostgreSQL.
        # For SQLite demonstration:
        self.db_connection = sqlite3.connect(self.config.db_file)
        cursor = self.db_connection.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS user_memories (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
```

```
            user_id TEXT NOT NULL,
            memory_content TEXT NOT NULL,
            memory_type TEXT NOT NULL,
            importance_score REAL DEFAULT 0.5,
            created_at TEXT DEFAULT CURRENT_TIMESTAMP,
            updated_at TEXT DEFAULT CURRENT_TIMESTAMP,
            encrypted_data BLOB,
            metadata JSON
          );
        """)
        self.db_connection.commit()
        self.logger.info(f"Database schema initialized in {self.config.db_file}.")

    async def store_memory(self, user_id: str, memory: Dict[str, Any]) -> bool:
        """
        Stores a memory with production safeguards including input validation,
        user limits, encryption, and metrics tracking.
        """
        start_time = asyncio.get_event_loop().time()
        success = False
        try:
          # Validate input for security and correctness.
          if not self._validate_memory_input(user_id, memory):
            return False

          # Check and enforce user memory limits.
          if not await self._check_user_limits(user_id):
            self.logger.warning(f"User {user_id} exceeded memory limit. Initiating cleanup.")
            await self._cleanup_old_memories(user_id) # Clean up to make room.

          # Encrypt sensitive data if configured.
          if self.config.enable_encryption:
            memory = self._encrypt_memory(memory) # Encryption is synchronous for this
demo.

          # Store in the persistent database.
          success = await self._store_in_database(user_id, memory)

          # Update the cache if storage was successful.
          if self.redis_client and success:
            await self._update_cache(user_id, memory)

          self.operation_counts['writes'] += 1
```

```
            return success

        except Exception as e:
            self.logger.error(f"Error storing memory for user {user_id}: {e}")
            self.operation_counts['errors'] += 1
            return False
        finally:
            execution_time = (asyncio.get_event_loop().time() - start_time) * 1000
            await self._record_operation('write', execution_time, success) # Record metrics
regardless of success.

    async def retrieve_memories(self, user_id: str, query: str, limit: int = 10) -> List]:
        """
        Retrieves memories, prioritizing cache and applying decryption if necessary.
        """
        start_time = asyncio.get_event_loop().time()
        memories: List] =
        success = False
        try:
            # Try to retrieve from cache first.
            cache_key = self._generate_cache_key(user_id, query, limit)
            cached_result = await self._get_from_cache(cache_key)

            if cached_result:
                self.operation_counts['cache_hits'] += 1
                memories = cached_result
                success = True
            else:
                self.operation_counts['cache_misses'] += 1
                # If not in cache, query the persistent database.
                memories = await self._query_database(user_id, query, limit)

                # Decrypt memories if encryption is enabled.
                if self.config.enable_encryption:
                    memories = self._decrypt_memories(memories) # Decryption is synchronous for
this demo.

                # Cache results for future requests.
                if self.redis_client:
                    await self._cache_results(cache_key, memories)
                success = True

            self.operation_counts['reads'] += 1
```

```
            return memories

        except Exception as e:
            self.logger.error(f"Error retrieving memories for user {user_id}: {e}")
            self.operation_counts['errors'] += 1
            return
        finally:
            execution_time = (asyncio.get_event_loop().time() - start_time) * 1000
            await self._record_operation('read', execution_time, success)

    def _validate_memory_input(self, user_id: str, memory: Dict[str, Any]) -> bool:
        """
        Validates memory input for security, correctness, and size limits.
        """
        if not user_id or not isinstance(memory, dict):
            self.logger.warning("Invalid user_id or memory format.")
            return False

        required_fields = ['content', 'type']
        if not all(field in memory for field in required_fields):
            self.logger.warning(f"Missing required fields in memory for user {user_id}.")
            return False

        content = memory.get('content', '')
        if len(content.encode('utf-8')) > self.config.max_memory_size_bytes:
            self.logger.warning(f"Memory content too large ({len(content.encode('utf-8'))} bytes)
for user {user_id}.")
            return False

        # Basic malicious content detection to prevent injection attacks.
        if self._contains_malicious_content(content):
            self.logger.warning(f"Malicious content detected in memory for user {user_id}.")
            return False

        return True

    def _contains_malicious_content(self, content: str) -> bool:
        """
        Performs a basic check for common malicious patterns (e.g., SQL injection, XSS).
        """
        malicious_patterns =
        content_lower = content.lower()
        return any(pattern in content_lower for pattern in malicious_patterns)
```

```
    async def _check_user_limits(self, user_id: str) -> bool:
        """
        Checks if a user is within their configured memory limits.
        Uses Redis for fast count checks if available.
        """
        try:
            count_key = f"memory_count:{user_id}"
            if self.redis_client:
                count_str = await self.redis_client.get(count_key)
                count = int(count_str) if count_str else 0
                if count >= self.config.max_memory_per_user:
                    self.logger.info(f"User {user_id} has {count} memories, exceeding limit of
{self.config.max_memory_per_user}.")
                    return False
                await self.redis_client.incr(count_key) # Increment count for new memory
            # In a real DB, you'd query the DB for count.
            return True
        except Exception as e:
            self.logger.error(f"Error checking user limits for {user_id}: {e}. Failing open.")
            return True # Fail open if there's an error with the limit check.

    async def _cleanup_old_memories(self, user_id: str):
        """
        Initiates cleanup of old memories for a user to manage storage space.
        This would involve deleting from the database and updating cache.
        """
        try:
            cutoff_date = datetime.now() - timedelta(days=self.config.memory_ttl_days)
            # In production, this would execute SQL DELETE statements.
            # For demo, simulate deletion.
            self.logger.info(f"Simulating cleanup of memories older than {cutoff_date.isoformat()}
for user {user_id}.")
            if self.redis_client:
                await self.redis_client.delete(f"memory_count:{user_id}") # Reset count or
decrement accurately.
                await self.redis_client.delete(f"user_memories:{user_id}:*") # Clear user's cache.
            self.logger.info(f"Cleaned up old memories for user {user_id}.")
        except Exception as e:
            self.logger.error(f"Error cleaning up memories for user {user_id}: {e}")

    def _encrypt_memory(self, memory: Dict[str, Any]) -> Dict[str, Any]:
        """
```

```
        Encrypts sensitive memory content.
        Simplified encryption for demonstration; use robust libraries (e.g., `cryptography`) in production.
        """
        content = memory.get('content', '')
        # For demonstration, use a simple hash as "encryption".
        # In production, use `cryptography.fernet` or similar.[21, 22]
        encrypted_content = hashlib.sha256(content.encode()).hexdigest()
        memory['encrypted_content'] = encrypted_content
        memory['is_encrypted'] = True
        # Remove original content if only encrypted version should be stored.
        memory.pop('content', None)
        return memory

    def _decrypt_memories(self, memories: List]) -> List]:
        """
        Decrypts sensitive memory content.
        """
        decrypted_list =
        for memory in memories:
            if memory.get('is_encrypted') and 'encrypted_content' in memory:
                # This is a placeholder; actual decryption would happen here.
                # Since we used SHA256 (one-way hash) for encryption in _encrypt_memory,
                # this cannot be truly decrypted
```

Works cited

1. medium.com, accessed on July 12, 2025, https://medium.com/%40nomannayeem/building-ai-agents-that-actually-remember-a-developers-guide-to-memory-management-in-2025-062fd0be80a1
2. Changelog - OpenAI API, accessed on July 12, 2025, https://platform.openai.com/docs/changelog
3. Migrating off ConversationBufferMemory or ConversationStringBufferMemory | 🦜 LangChain, accessed on July 12, 2025, https://python.langchain.com/docs/versions/migrating_memory/conversation_buffer_memory/
4. LangChain Memory Component Deep Dive: Chain Components and Runnable Study, accessed on July 12, 2025, https://dev.to/jamesli/langchain-memory-component-deep-dive-chain-components-and-runnable-study-359p
5. No clear v2 option for ConversationChain with ConversationSummaryBufferMemory, as RunnableWithMessageHistory has critical gap #24562 - GitHub, accessed on July 12, 2025, https://github.com/langchain-ai/langchain/discussions/24562

6. How to pass ConversationBufferWindowMemory with LCEL chain #22553 - GitHub, accessed on July 12, 2025, https://github.com/langchain-ai/langchain/discussions/22553
7. Chat models - Python LangChain, accessed on July 12, 2025, https://python.langchain.com/docs/concepts/chat_models/
8. Class ChatOpenAI
9. ConversationSummaryMemory — LangChain documentation, accessed on July 12, 2025, https://python.langchain.com/api_reference/langchain/memory/langchain.memory.summary.ConversationSummaryMemory.html
10. Pydantic AI Tracing | Arize Docs, accessed on July 12, 2025, https://arize.com/docs/ax/integrations/pydantic/pydantic-ai-tracing
11. llms-full.txt - Pydantic AI, accessed on July 12, 2025, https://ai.pydantic.dev/llms-full.txt
12. Messages and chat history - PydanticAI, accessed on July 12, 2025, https://ai.pydantic.dev/message-history/
13. AI Agents X : Agno — Agentic Framework | by DhanushKumar | Jun, 2025 - Medium, accessed on July 12, 2025, https://medium.com/@danushidk507/ai-agents-x-agno-agentic-framework-2a2abba49604
14. ConversationEntityMemory — LangChain documentation, accessed on July 12, 2025, https://python.langchain.com/api_reference/langchain/memory/langchain.memory.entity.ConversationEntityMemory.html
15. Agno: The agent framework for Python teams - WorkOS, accessed on July 12, 2025, https://workos.com/blog/agno-the-agent-framework-for-python-teams
16. Building an AI Agent with Agno: A Step-by-Step Guide - Artificial Intelligence in Plain English, accessed on July 12, 2025, https://ai.plainenglish.io/building-an-ai-agent-with-agno-a-step-by-step-guide-13542b2a5fb6
17. langchain.memory.combined.CombinedMemory, accessed on July 12, 2025, https://api.python.langchain.com/en/latest/memory/langchain.memory.combined.CombinedMemory.html
18. redis-async-client - PyPI, accessed on July 12, 2025, https://pypi.org/project/redis-async-client/
19. Async performance issue since version 5.3.0 · Issue #3692 · redis/redis-py - GitHub, accessed on July 12, 2025, https://github.com/redis/redis-py/issues/3692
20. Async Caching in Python with aiocache and Redis | by PI | Neural Engineer - Medium, accessed on July 12, 2025, https://medium.com/neural-engineer/async-caching-in-python-with-aiocache-and-redis-8ca985614a9a
21. What is Python Fernet ? How to use it in 2022? - HideIPVPN, accessed on July 12, 2025, https://www.hideipvpn.com/privacy/what-is-python-fernet/
22. fernet - NPM, accessed on July 12, 2025, https://www.npmjs.com/package/fernet