# Chapter 6. Knowledge and Memory

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the sixth chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *sevans@oreilly.com*.

Now that your agent has skills and orchestration, it is more than capable of taking actions to do real work. In most cases, though, you will want your agents to know more about your specific problem area than the foundation model would alone, and be able to store knowledge and information over time. You might even want your agent to use this memory to accomplish tasks more effectively. In this chapter, we'll discuss how memory can be added to agentic systems to add external knowledge, maintain state across sessions, and perform complex tasks more effectively. Memory can also be an excellent way to get your AI-powered application to better meet the needs of the specific context. Let's dive in.

Memory plays multiple roles in applications with AI Agents. In this context, we are not referring to computer system memory, such as RAM, but to information that is dynamically injected into the prompt to complement the parametric memory of the model. It maintains the state of the interaction, the previous tasks performed and their previous results, and is critical for learning. These approaches are useful for the interactions between humans and LLM-powered applications, the interactions between AI agents, and for domain-specific or organization-specific information. We'll discuss specific architectures soon, but first, we'll focus on the fundamentals of memory for agentic systems.

## Foundational Approaches to Memory

We begin by discussing the simplest approaches to memory: relying on a rolling context window for the foundation model, and keyword-based memory. Despite their simplicity, they are more than sufficient for a wide range of use cases.

### Managing Context Windows

We start with the simplest approach to memory: relying on the context window. The context window is a critical resource for developers to use effectively. We want to provide the foundation model with all the infor-

mation it needs to complex the task, but no more. The context window is all of the information that is provided to the foundation model when the model is called. In the simplest approach, in addition to the current question, all of the remaining context window that is available is filled with the previous interactions in the current session. When that window fills up, only the most recent interactions are included. In some circumstances, we will have more information to provide than we can fit into the context window. When this happens, we need to be careful with how we allocate our limited budget of tokens.

For simple use cases, you can use a rolling context window. In this case, as the interaction with the foundation model progresses, the full interaction is passed into the context window. At a certain point, the context window fills up, the oldest parts of the context are ejected, and replaced with the most recent context, in a first-in, first-out fashion. This is easy to implement, low in complexity, and will work for many use cases. The primary drawback to this approach is information will be lost, regardless of how relevant or important it is, as soon as enough interaction has occurred to eject it from the current context. With large prompts or verbose foundation model responses, this can happen quickly. Foundation models can also miss important information in large prompts, so highlighting the most relevant context, and placing it close to the end of the prompt can increase the likelihood that it will be used. This standard approach to memory can be incorporated into our LangGraph agent as follows:

```
from typing import Annotated
from typing_extensions import TypedDict

from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, MessagesState, START

llm = ChatOpenAI(model="gpt-4o")


def call_model(state: MessagesState):
   response = llm.invoke(state["messages"])
   return {"messages": response}


builder = StateGraph(MessagesState)
builder.add_node("call_model", call_model)
builder.add_edge(START, "call_model")
graph = builder.compile()

# Fails to maintain state across the conversation
input_message = {"type": "user", "content": "hi! I'm bob"}
for chunk in graph.stream({"messages": [input_message]}, stream_mode="values"):
   chunk["messages"][-1].pretty_print()

input_message = {"type": "user", "content": "what's my name?"}
for chunk in graph.stream({"messages": [input_message]}, stream_mode="values"):
   chunk["messages"][-1].pretty_print()
```

## Keyword-Based Memory

The simplest way to begin extracting and organizing larger sessions as they go is through keyword extraction. One task that foundation models have performed very well at is keyword extraction. Given an input block of text, a foundation model can identify the key words and phrases in it.

As the interaction occurs, send each response to a foundation model with a keyword extraction prompt. Maintain storage for each response, as well as a map from the keywords to their original document. When new prompts are received, keywords are extracted from it, and the lookup is consulted. If there are matches, the previous occurrences are included in the prompt. This simple approach can preserve the broader context of interactions that address specific topics over time. An advantage to this approach is its simplicity, while still enabling the preservation of information over time. Some important parameters to choose for this type of memory are

### Integrating Keyword-Based Memory with Rolling Context Windows

While both rolling context windows and keyword-based memory have their own advantages, we can also take a hybrid approach that splits the context window between keyword retrieval and rolling context window. Include the j most recent occurrences from the keyword retrieval, then fill the rest of the context window with the k most recent interactions that fit into the context window. But what if we want to include relevant interactions, even if they don't have a keyword match, or occurred far enough back that it wouldn't be in the context window? We'll address that in the next section.

## Semantic Memory and Vector Stores

Semantic memory, a type of long-term memory that involves the storage and retrieval of general knowledge, concepts and past experiences, plays a critical role in enhancing the cognitive capabilities of these systems. This allows information and past experiences to be stored and then efficiently retrieved when it is needed to improve performance later on. The leading way to do this is by using vector databases, which enable rapid indexing and retrieval at large scale, enabling agentic systems to understand and respond to queries with greater depth and relevance.

### Introduction to Semantic Search

Unlike traditional keyword-based search, semantic search aims to understand the context and intent behind a query, leading to more accurate and meaningful retrieval results. At its core, semantic search focuses on the meaning of words and phrases rather than their exact match. It leverages machine learning techniques to interpret the context, synonyms, and relationships between words. This allows the retrieval system to comprehend the intention and deliver results that are contextually relevant, even if they don't contain the exact search terms.

The foundation for these approaches are embeddings. These are vector representations of words that capture their meanings based on their usage in large text corpora. By projecting large bodies of text into a dense numeric representation, we can create rich representations that have proven to be very useful for storage and retrieval. Popular models like Word2Vec, GloVe, and BERT have revolutionized how machines understand language by placing semantically similar words closer together in a high-dimensional space. Large language models have further improved the performance of these embedding models across a wide range of types

of text by increasing the size of the embedding model and the quantity and variety of data on which they are trained. Semantic search has proven to be an invaluable technique to improve the performance of memory within agentic systems, particularly in retrieving semantically relevant information across documents that do not share exact keywords.

## Implementing Semantic Memory with Vector Stores

We begin by generating semantic embeddings for the concepts and knowledge to be stored. These embeddings are typically produced by large language models or other NLP techniques that encode textual information into dense vector representations. These vector representations, or embeddings, capture the semantic properties and relationships of data points in a continuous vector space. For example, a sentence describing a historical event can be converted into a vector that captures its semantic meaning. Once we have this vector representation, we need a place to efficiently store it. That palace is a vector database, which are designed specifically to efficiently handle high-dimensional vector representations of data.

Vector stores, such as vectordb, FAISS (Facebook AI Similarity Search) or Annoy (Approximate Nearest Neighbors Oh Yeah), are optimized for storing and searching high-dimensional vectors. These stores allow for fast similarity searches, enabling the retrieval of embeddings that are semantically similar to a given query.

When an agent receives a query or needs to retrieve information, it can use the vector store to perform similarity searches based on the query's embedding. By finding and retrieving the most relevant embeddings from the vector store, the agent can access the stored semantic memory and provide informed, contextually appropriate responses. These lookups can be performed quickly, providing an efficient way to rapidly search over large volumes of information to improve the quality of actions and responses. This can be implemented as follows:

```
from typing import Annotated
from typing_extensions import TypedDict

from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, MessagesState, START

llm = ChatOpenAI(model="gpt-4o")

def call_model(state: MessagesState):
   response = llm.invoke(state["messages"])
   return {"messages": response}

from vectordb import Memory

memory = Memory(chunking_strategy={'mode':'sliding_window', 'window_size': 128, 'overlap': 16})

text = """
Machine learning is a method of data analysis that automates analytical model building.

It is a branch of artificial intelligence based on the idea that systems can learn from data,
identify patterns and make decisions with minimal human intervention.
```

```
Machine learning algorithms are trained on data sets that contain examples of the desired output. For ex
Once an algorithm is trained, it can be used to make predictions on new data. For example, the machine l
"""

metadata = {"title": "Introduction to Machine Learning", "url": "https://example.com/introduction-to-mac

memory.save(text, metadata)

text2 = """
Artificial intelligence (AI) is the simulation of human intelligence in machines
that are programmed to think like humans and mimic their actions.

The term may also be applied to any machine that exhibits traits associated with
a human mind such as learning and problem-solving.

AI research has been highly successful in developing effective techniques for solving a wide range of pr
"""

metadata2 = {"title": "Introduction to Artificial Intelligence", "url": "https://example.com/introductio

memory.save(text2, metadata2)

query = "What is the relationship between AI and machine learning?"

results = memory.search(query, top_n=3)

builder = StateGraph(MessagesState)
builder.add_node("call_model", call_model)
builder.add_edge(START, "call_model")
graph = builder.compile()

input_message = {"type": "user", "content": "hi! I'm bob"}
for chunk in graph.stream({"messages": [input_message]}, {}, stream_mode="values"):
   chunk["messages"][-1].pretty_print()

print(results)
```

## Retrieval Augmented Generation

Incorporating memory into agentic systems not only involves storing and managing knowledge but also enhancing the system's ability to generate contextually relevant and accurate responses. Retrieval Augmented Generation (RAG) is a powerful technique that combines the strengths of retrieval-based methods and generative models to achieve this goal. By integrating retrieval mechanisms with large language models, RAG allows agentic systems to generate more informed and contextually enriched responses, improving their performance in a wide range of applications.

During retrieval, the system searches a large corpus of documents or a vector store of embeddings to find pieces of information that are relevant to the given query or context. This phase relies on efficient retrieval mechanisms to quickly identify and extract pertinent information.

During generation, the retrieved information is then fed into a generative foundation model, which uses this context to produce a coherent and contextually appropriate response. The generative model synthesizes the retrieved data with its own learned knowledge, enhancing the relevance and accuracy of the generated text.

Retrieval Augmented Generation represents a powerful approach for enhancing the capabilities of agentic systems by combining retrieval-based

methods with generative models. By leveraging external knowledge and integrating it into the generation process, RAG enables the creation of more informed, accurate, and contextually relevant responses. As technology continues to evolve, RAG will play a crucial role in advancing the performance and versatility of LLM-powered applications across various domains. This is especially valuable for incorporating domain- or company- specific information or policies to influence the output.

### Semantic Experience Memory

While incorporating an external knowledge base with a semantic store is an effective way to incorporate external knowledge into our agent, our agent will start every session from a blank slate, and the context of long-running or complex tasks will gradually drop out of the context window. Both of these issues can be addressed by semantic experience memory.

With each user input, the text is turned into a vector representation using an embedding model. The embedding is then used as the query in a vector search across all of the previous interactions in the memory store. Part of the context window is reserved for the best matches from the semantic experience memory, then the rest of the space is allocated to the system message, latest user input, and the most recent interactions. Semantic experience memory allows agentic systems to not only draw upon a broad base of knowledge but also tailor their responses and actions based on accumulated experience, leading to more adaptive and personalized behavior.

## Graph RAG

We now turn to an advanced version of RAG that is more complex to incorporate into your solution, but that is capable of correctly handling a wider variety of questions. Graph Retrieval Augmented Generation (Graph RAG) is an advanced extension of the Retrieval Augmented Generation (RAG) model, incorporating graph-based data structures to enhance the retrieval process. By utilizing graphs, Graph RAG can manage and utilize complex interrelationships and dependencies between pieces of information, significantly enhancing the richness and accuracy of the generated content. This chapter will delve into how Graph RAG works, its implementation, and its applications in various domains.

### Using Knowledge Graphs

Graph RAG extends the basic RAG framework by integrating a graph-based retrieval system. This system leverages the power of graph databases or knowledge graphs to store and query interconnected data. In Graph RAG, the retrieval phase doesn't just pull relevant documents or snippets; it analyzes and retrieves nodes and edges from a graph that represent complex relationships and contexts within the data. GraphRAG consists of the following three components:

*Knowledge Graph*

This component stores data in a graph format, where entities (nodes) and their relationships (edges) are explicitly defined. Graph databases are highly efficient at managing connected data and sup-

porting complex queries that involve multiple hops or relationships.

*Retrieval System*

The retrieval system in Graph RAG is designed to query the graph database efficiently, extracting subgraphs or clusters of nodes that are most relevant to the input query or context.

*Generative Model*

Once relevant data is retrieved in the form of a graph, the generative model synthesizes this information to create coherent and contextually rich responses.

Graph Retrieval-Augmented Generation represents a significant leap forward in the capabilities of agentic systems, offering sophisticated tools to handle and generate responses based on complex interconnected data. As this technology evolves, it promises to open new frontiers in AI applications, making systems smarter, more context-aware, and capable of handling increasingly complex tasks. Using knowledge graphs in Graph RAG systems transforms the way information is retrieved and utilized for generation, enabling more intelligent, contextual, and accurate responses across various applications. We will not cover the details of the algorithm here, but multiple open-source implementations of GraphRAG are now available, and setting them up on your dataset is easier to do. If you have a large set of data you need to reason over, and standard chunking with a vector retrieval is running into limitations, GraphRAG is a more expensive and complex approach that frequently produces better results in practice.

## Building Knowledge Graphs

Knowledge graphs are fundamental in providing structured and semantically rich information that enhances the capabilities of intelligent systems, including Graph Retrieval-Augmented Generation (Graph RAG) systems. Building an effective knowledge graph involves a series of steps, from data collection and processing to integration and maintenance. This section will cover the methodology for constructing knowledge graphs that can significantly impact the performance of Graph RAG systems. This process consists of several steps:

1. **Data Collection**: The first step in building a knowledge graph is gathering the necessary data. This data can come from various sources, including databases, text documents, websites, and even user-generated content. It's crucial to ensure the diversity and quality of sources to cover a broad spectrum of knowledge. For an organization, this may consist of a set of core policies or documents that contain core information to influence the agent.
2. **Data Preprocessing**: Once data is collected, it needs to be cleaned and preprocessed. This step involves removing irrelevant or redundant information, correcting errors, and standardizing data formats. Preprocessing is vital for reducing noise in the data and improving the accuracy of the subsequent entity extraction process.
3. **Entity Recognition and Extraction**: This process involves identifying key elements (entities) from the data that will serve as nodes in the knowledge graph. Common entities include people, places, organizations, and concepts. Techniques such as Named Entity Recognition

(NER) are typically used, which may involve machine learning models trained on large datasets to recognize and categorize entities accurately.

4. **Relationship Extraction**: After identifying entities, the next step is to determine the relationships between them. This involves parsing data to extract predicates that connect entities, forming the edges of the graph. Relationship extraction can be challenging, especially in unstructured data, though foundation models have shown improving efficacy over time.
5. **Ontology Design**: An ontology defines the categories and relationships within the knowledge graph, serving as its backbone. Designing an ontology involves defining a schema that encapsulates the types of entities and the possible types of relationships between them. This schema helps in organizing the knowledge graph systematically and supports more effective querying and data retrieval.
6. **Graph Population**: With the ontology in place, the next step is to populate the graph with the extracted entities and their relationships. This involves creating nodes and edges in the graph database according to the ontology's structure. Databases like Neo4j, OrientDB, or Amazon Neptune can be used to manage these data structures efficiently.
7. **Integration and Validation**: Once the graph is populated, it must be integrated with existing systems and validated to ensure accuracy and utility. This can involve linking data from other databases, resolving entity duplication (entity resolution), and verifying that the graph accurately represents the knowledge domain. Validation might involve user testing or automated checks to ensure the integrity and usability of the graph.
8. **Maintenance and Updates**: A knowledge graph is not a static entity; it needs regular updates and maintenance to stay relevant. This involves adding new data, updating existing information, and refining the ontology as new types of entities or relationships are identified. Automation and machine learning models can be instrumental in maintaining and updating the knowledge graph efficiently.

Building a knowledge graph is a complex but rewarding endeavor that can significantly enhance the capabilities of Graph RAG systems. By structuring information into an interconnected web of knowledge, these graphs enable intelligent systems to perform sophisticated reasoning, provide contextual responses, and deliver personalized services. These structures make it easy to discover underlying relationships in the data. For instance, it is now possible to search for elements on the graph, then retrieve all the elements that are one or more links away from that node. This provides an efficient way to retrieve relevant context for addressing a task. As AI technology progresses, the methodologies for building, integrating, and maintaining knowledge graphs will continue to evolve, further enhancing their utility in various domains.

## Promise and Peril of Dynamic Knowledge Graphs

Dynamic knowledge graphs represent an evolutionary leap in managing and utilizing knowledge in real-time applications. These graphs are continuously updated with new information, adapting to changes in knowledge and context, which can significantly enhance Graph Retrieval-Augmented Generation (Graph RAG) systems. However, the dynamic nature of these graphs also introduces specific challenges that need careful consideration. This section explores the potential benefits and risks asso-

ciated with dynamic knowledge graphs. As the developer, it is important to apply careful consideration to what should be included in the knowledge graph. Official documentation, publicly available content, and past interactions could all be considered for knowledge graph construction, but it is important to remember that the graph construction process is imperfect, and poor quality or security vulnerabilities in the underlying data can undermine the system.

Dynamic Real-time information processing is greatly enhanced by dynamic knowledge graphs, which can integrate real-time data. This capability is particularly useful in environments where information is constantly changing, such as news, social media, and live monitoring systems. By ensuring that the system's responses are always based on the most current and relevant information, dynamic knowledge graphs provide a significant advantage.

Adaptive learning is another key feature of dynamic knowledge graphs. They continuously update themselves, learning from new data without the need for periodic retraining or manual updates. This adaptability is crucial for applications in fast-evolving fields like medicine, technology, and finance, where staying updated with the latest knowledge is critical. This helps organizations make informed decisions quickly, which is invaluable in scenarios where decisions have significant implications and depend heavily on the latest information. Knowledge graphs also provide critical information in a structured format that can be effectively operated across and reasoned over, and provide far greater flexibility than vector stores, and are especially valuable for understanding the rich context of an entity. Unfortunately, these benefits come with some important drawbacks:

*Complexity in Maintenance*

Maintaining the accuracy and reliability of a dynamic knowledge graph is significantly more challenging than managing a static one. The continuous influx of new data can introduce errors and inconsistencies, which may propagate through the graph if not identified and corrected promptly.

*Resource Intensity*

The processes of updating, validating, and maintaining dynamic knowledge graphs require substantial computational resources. These processes can become resource-intensive, especially as the size and complexity of the graph grow, potentially limiting scalability.

*Security and Privacy Concerns*

Dynamic knowledge graphs that incorporate user data or sensitive information must be managed with strict adherence to security and privacy standards. The real-time aspect of these graphs can complicate compliance with data protection regulations, as any oversight might lead to significant breaches.

*Dependency and Overreliance*

There is a risk of overreliance on dynamic knowledge graphs for decision-making, potentially leading to a lack of critical oversight.

Decisions driven solely by automated insights from a graph might overlook external factors that the graph does not capture.

To harness the benefits of dynamic knowledge graphs while mitigating their risks, several strategies can be employed. Implementing robust validation mechanisms with automated tools and processes is essential for continuously ensuring the accuracy and reliability of data within the graph. Designing a scalable architecture using technologies such as distributed databases and cloud computing helps manage the computational demands of dynamic graphs. Strong security measures, including encryption, access controls, and anonymization techniques, are crucial to ensure that all data inputs and integrations comply with current security and privacy regulations. Additionally, maintaining human oversight in critical decision-making processes mitigates the risks of errors and overreliance on automated systems.

Dynamic knowledge graphs offer substantial promise for enhancing the intelligence and responsiveness of Graph RAG systems, providing significant benefits across various applications. However, the complexities and risks associated with their dynamic nature necessitate careful management and oversight. By addressing these challenges proactively, the potential of dynamic knowledge graphs can be fully realized, driving forward the capabilities of intelligent systems in an ever-evolving digital landscape.

# Working Memory

Working memory plays a crucial role similar to short-term memory in human cognition. It enables agents to hold and manipulate information temporarily for the execution of complex tasks and interactions. This chapter explores the concept of working memory in agentic systems, discussing its functions, implementations, and the challenges associated with integrating it into large language models and other intelligent systems.

Working memory in agentic systems refers to the temporary storage and processing space used to hold immediate data and contextual information needed for task execution. This type of memory is dynamic, rapidly updateable, and crucial for tasks that require comprehension, reasoning, and immediate response, such as conversational understanding, problem-solving, and dynamic decision-making.

## Whiteboards

Just as for humans, it is not always possible to solve problems directly. Sometimes, we need to work through a problem step by step, and save our notes as we go. This is exactly what a “whiteboard” is for a foundation model - it serves as a dynamic form of working memory.[1] Conceptually similar to the physical whiteboards used in brainstorming sessions, digital whiteboards in agentic systems provide a flexible, interactive canvas where temporary data can be stored, manipulated, and used collaboratively. This section explores how whiteboards function as an integral part of working memory in intelligent agents, their implementations, and their applications in enhancing system performance.

Whiteboards in agentic systems function as a short-term storage, where multiple streams of data and processes can be dynamically displayed and managed. Agents can interact with and manipulate the data on whiteboards, allowing for dynamic changes based on new inputs or decisions. These systems have also been called scratchpads[2]. This feature is crucial for tasks that require real-time adjustments or updates. It also allows for important information to be extracted and made readily available for reference, which can be incredibly useful when operating over large amounts of data. These have been shown to be especially useful for complex and multi-step computations, as well as predicting outputs from arbitrary programs.

### Note Taking

While closely related to whiteboards, note taking is a distinct approach that can also improve performance for complex tasks. With this technique, the foundation model is prompted to specifically inject notes on the input context without trying to answer the question.[3] This mimics the way that we might fill in the margins or summarize a paragraph or section. This note-taking is performed before the question is presented, and then interleaves these notes with the original context when attempting to address the current task. Experiments show good results on multiple reasoning and evaluation tasks, with potential for adaptation to a wider range of scenarios. As we can see in the figure below, in a traditional, vanilla approach, the model is provided with the context and a question, and produces an answer. In chain of thought, it has time to reason about the problem, and only subsequently generate its answer to the question. With the self-note approach, the model generates notes on multiple parts of the context, and then generates a note on the question, before finally moving to generate the final answer.

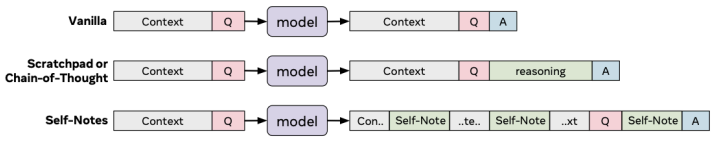


Figure 6-1. Taking Notes on the Context and Question to Improving Answer Quality (Source: *https://arxiv.org/pdf/2305.00833*)

# Conclusion

Memory is critical to the successful operation of agentic systems, and while the standard approach of relying on the context window of recent interactions is sufficient for many use cases, more challenging scenarios can benefit substantially from the investment into a more robust approach. We have explored several approaches here, including semantic memory, GraphRAG, and working memory.

This chapter on memory in agentic systems has delved into various aspects of how memory can be structured and utilized to enhance the capabilities of intelligent agents. From the basic concepts of managing context windows, through the advanced applications of semantic memory and vector stores, to the innovative practices of dynamic knowledge graphs and working memory, we have explored a comprehensive range of techniques and technologies that play crucial roles in the development of agentic systems.

Memory systems in agentic applications are not just about storing data but about transforming how agents interact with their environment and end-users. By continually improving these systems, we can create more intelligent, responsive, and capable agents that can perform a wide range of tasks more effectively. In the next chapter, we will explore how agents can learn from experience to improve automatically over time.

[1] Whiteboard of Thought: Thinking Step-by-Step Across Modalities, *https://arxiv.org/pdf/2406.14562*

[2] Show Your Work: Scratchpads for Intermediate Computation with Language Models, *https://arxiv.org/pdf/2112.00114*

[3] Learning to Reason with Self-Notes, *https://arxiv.org/abs/2305.00833*