Chapter 6. Knowledge and Memory

Now that your agent has tools and orchestration, it is more than capable of taking actions to do real work. In most cases, though, you will want your agents to both remember what's happened and know additional information beyond what lives in the model's weights. In this chapter, we'll focus on knowledge and memory—two complementary but distinct ways to enrich your agent's context. Knowledge (often implemented via retrieval-augmented generation) pulls in factual or domain-specific content—technical specs, policy documents, product catalogs, customer or system logs—at generation time so the agent "knows" verifiable information beyond the immediate conversation to complement the information stored in the model itself, specifically in its weights and biases. Memory, on the other hand, captures the agent's own history: prior user exchanges, tool outputs, and state updates. It lets your agent maintain continuity across turns and sessions so that it "remembers" past interactions and uses that history to inform future decisions.

In Chapter 5, we introduced context engineering as the discipline of dynamically selecting, structuring, and assembling all inputs into the model's context window to produce the best outcomes. Memory is a foundational enabler of context engineering: it provides the knowledge, history, and facts that can be selected and assembled into effective prompts. In other words, memory is where knowledge is stored, while context engineering is how that knowledge is leveraged to produce intelligent behavior.

This chapter will offer examples in LangGraph, a low-level orchestration framework for building stateful agentic workflows that was introduced in Chapter 1. LangGraph defines your application as a directed graph of nodes (pure functions such as foundation model calls, memory updates, or tool invocations) and edges (control-flow transitions), enabling developers to model complex, multistep processes declaratively. LangGraph treats your entire application state as a single, strongly typed Python object (often a TypedDict) that flows through the graph at runtime, keeping data management both explicit and type-safe. Unlike DAG-only (directed acyclic graph) orchestration tools, it natively supports cycles and conditional branches,

making it straightforward to implement loops, retries, and dynamic decision paths without bespoke code. It also provides built-in streaming—emitting partial outcomes as they are generated—and checkpointing, so long-running agents can persist and resume exactly where they left off.

By treating memory mechanisms (rolling context windows, keyword extraction, semantic retrieval, etc.) as first-class graph nodes, LangGraph keeps memory logic modular and testable. Edges ensure memory updates occur in the correct sequence relative to LLM calls, so your agent always has the right context injected at the right time. And because state—including memory contents—can be checkpointed and resumed, your agents maintain continuity across sessions and withstand failures, all within the same unified graph framework.

In this chapter, we will first cover the fundamentals of memory for agentic systems, from simple rolling context windows to semantic memory, retrieval-augmented generation, and advanced knowledge graph approaches. Throughout, we will emphasize how these memory systems integrate into context engineering pipelines to build agents that are grounded, capable, and aligned with your specific goals and environment.

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” refers to the information that is passed to the foundation model as an input in a single call. The maximum number of tokens a foundational model can ingest and attend to in a single call is called the “context length.” This context is effectively the working memory for that request. One token averages about ¾ of a word or roughly four characters; for example, 1,000 tokens correspond to about 750 English words. Many popular models today have stepped through roughly 4,000-token (≈3,000 words, ~12 pages) and 8,000-token (≈6,000 words, ~24 pages) limits. GPT-5 and Claude

3.7 Sonnet now offer a maximum number of 272,000 tokens in their input, while Gemini 2.5 accepts up to a million tokens in the input.

The context window is a critical resource for developers to use effectively. We want to provide the foundation model with all the information it needs to complete 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, the context window contains the current question and all 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, and 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,

llm = ChatOpenAI(model="gpt-5")

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

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

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

Traditional Full-Text Search

Traditional full-text search forms the backbone of many large-scale retrieval systems and offers a robust, mature approach to injecting precise historical context into agents enabled with foundation models. At its heart lies an inverted index, which preprocesses all text via tokenization, normalization (lowercasing, stemming), and stop-word removal, then maps each term to the list of message chunks or documents in which it appears. This structure enables lightning-fast lookups—rather than scanning every stored message, the agent simply follows the term's postings list to retrieve exactly those passages containing the query keywords.

To rank these results by relevance, most systems employ the BM25 scoring function. BM25 weights each passage by its term frequency (how often the query term appears), inverse document frequency (how rare the term is across the corpus), and document length normalization (penalizing overly long or overly short chunks). When a user query arrives, it is analyzed with the same text pipeline used for indexing, and BM25 produces a sorted list of the top K candidate passages. These top hits—often truncated or summarized—are then injected directly into the foundation model prompt, ensuring the model sees the most pertinent historical context without exhausting its context length. Fortunately, implementing this is very easy to do in Python, though typically one would store these in a database:

```
# pip install rank_bm25

from rank_bm25 import BM25Okapi
from typing import List

corpus: List[List[str]] = [
    "Agent J is the fresh recruit with attitude".split(
```

```
    "Agent K has years of MIB experience and a cool neu
    "The galaxy is saved by two Agents in black suits".
]
# 2. Build the BM25 index
bm25 = BM25Okapi(corpus)

# 3. Perform retrieval for a fun query
query = "Who is a recruit?".split()
top_n = bm25.get_top_n(query, corpus, n=2)

print("Query:", " ".join(query))
print("Top matching lines:")
for line in top_n:
    print(" •", " ".join(line))
```

In this example, we built a simple BM25-powered full-text index over our agent quips and fetched the most relevant lines for a given user query. By injecting those top-ranked passages directly into the prompt, we ensure the model has the key historical context—without passing every past message—and stays within its context limits.

While this keyword-driven approach excels at pinpointing exact or highly specific terms, it can miss broader themes, paraphrases, or conceptual links that weren't expressed in the original text. To capture that deeper, "meaning-based" memory—so your agent can recall related ideas even when the exact words differ—we turn next to semantic memory and vector stores.

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 for information and past experiences to be stored and then efficiently retrieved when they are 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 ML techniques to interpret the context, synonyms, and relationships between words. This enables 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 is embeddings, which are vector representations of words that capture the words' 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 (LLMs) 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 foundation models or other natural language processing (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 place is a vector database, which is 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 are set up 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,
llm = ChatOpenAI(model="gpt-5")
def call_model(state: MessagesState):
   response = llm.invoke(state["messages"])
   return {"messages": response}
from vectordb import Memory
memory = Memory(chunking_strategy={'mode':'sliding_winc
'overlap': 16})
text = """
Machine learning is a method of data analysis that autc
model building. It is a branch of artificial intelliger
idea that systems can learn from data, identify patterr
decisions with minimal human intervention. Machine lear
are trained on datasets that contain examples of the de
For example, a machine learning algorithm that is used
images might be trained on a dataset that contains imag
and dogs. Once an algorithm is trained, it can be used
predictions on new data. For example, the machine learr
that is used to classify images could be used to predic
image contains a cat or a dog.
"""
metadata = {"title": "Introduction to Machine Learning"
"https://learn.microsoft.com/en-us/training/modules/" +
    "introduction-to-machine-learning"}
memory.save(text, metadata)
```

```
text2 = """
Artificial intelligence (AI) is the simulation of humar
that are programmed to think like humans and mimic thei
The term may also be applied to any machine that exhibi
a human mind such as learning and problem-solving.
AI research has been highly successful in developing ef
solving a wide range of problems, from game playing to
"""
metadata2 = {"title": "Artificial Intelligence for Begi
"https://microsoft.github.io/AI-for-Beginners"}
memory.save(text2, metadata2)
query = "What is the relationship between AI and machir
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 bc
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 foundation models, RAG enables agentic systems to generate more informed and contextually enriched responses, improving their performance in a wide range of applications.

First, we begin with a set of documents that might be useful to help the system answer questions. We then break these documents into smaller chunks. The idea is that the model, like a person, doesn't need to refer to an entire long resource—it only needs the small, relevant part. We then take these chunks, embed them with an encoder model, and index them in a vector database, as illustrated in Figure 6-1.

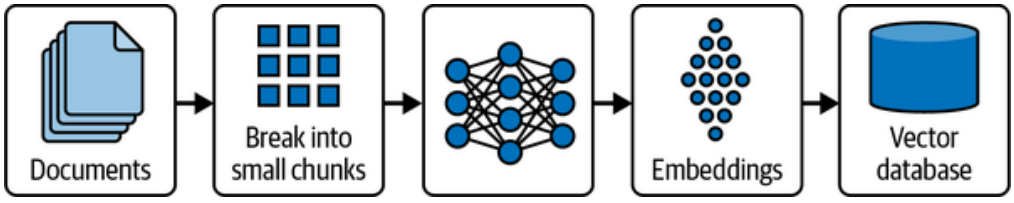

Figure 6-1. Indexing pipeline for RAG. Source documents are first split into smaller chunks. Each chunk is converted into a dense embedding by an encoder model, and the resulting vectors are stored in a vector database—enabling fast semantic lookup at query time.

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, as is illustrated in Figure 6-2.

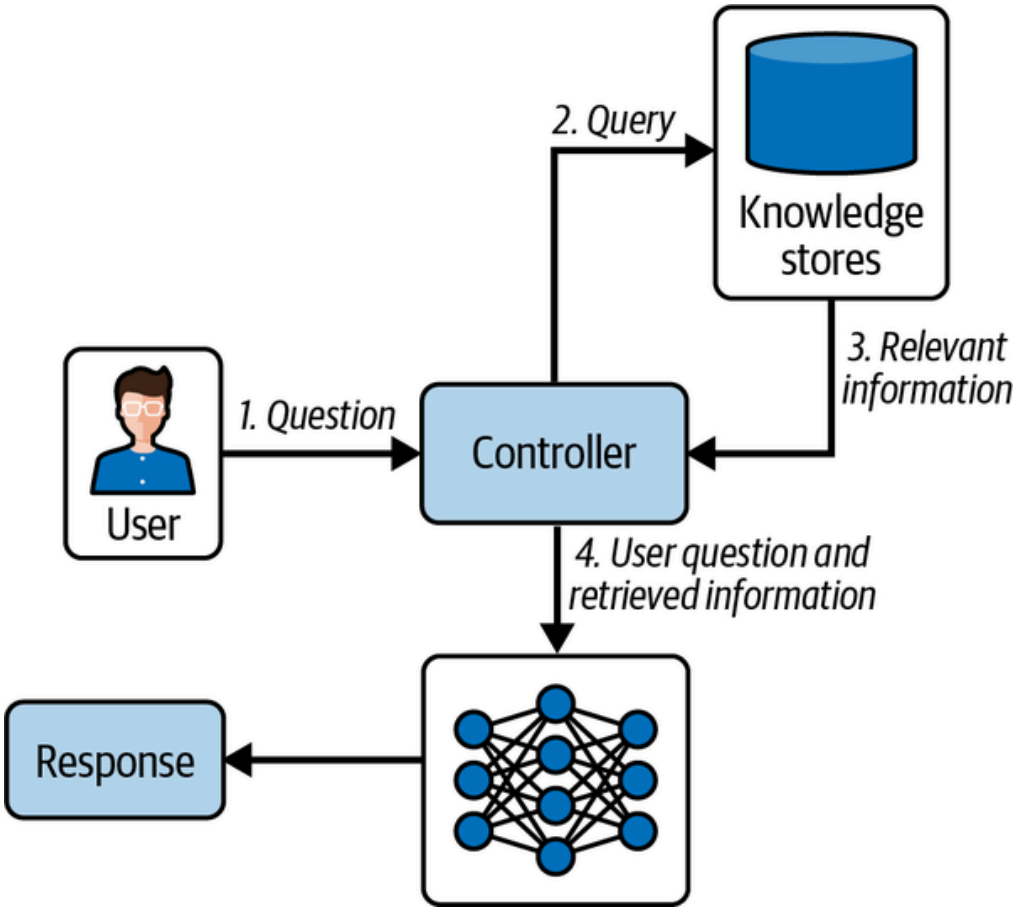

Figure 6-2. RAG runtime workflow. The user submits a question to the controller, which queries the vector knowledge store to retrieve the most relevant information. This retrieved context is then combined with the original user question and passed to the generative model, which produces a final, contextually informed response.

RAG 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 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.

GraphRAG

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 (GraphRAG) is an advanced extension of the RAG model, incorporating graph-based data structures to enhance the retrieval process. By utilizing graphs, GraphRAG can manage and utilize complex interrelationships and dependencies between pieces of information, significantly enhancing the richness and accuracy of the generated content.

Baseline RAG systems operate by chunking documents, embedding those chunks into vector space, and retrieving semantically similar chunks at query time to augment prompts for the LLM. While effective for simple fact lookup or direct question-answering, this approach struggles when:

- Answers require connecting information scattered across multiple documents (“connecting the dots”).
- Queries involve summarizing higher-level semantic themes across a dataset.
- The dataset is large, messy, or organized narratively rather than as discrete facts.

For example, baseline RAG might fail to answer “What has Geoffrey Hinton done?” if no single retrieved chunk covers his actions comprehensively. GraphRAG addresses this by constructing a knowledge graph of entities and relationships from the dataset, enabling multihop reasoning, relationship chaining, and structured summarization.

Using Knowledge Graphs

Within a few minutes, the GraphRAG CLI can deliver global insights and local context over your texts—no Python required. But if you want more control and flexibility, production-level pipelines are just a few lines away using the neo4j-graphrag-python package. With the official neo4j-graphrag library, setup involves only configuring a Neo4j connection, defining an embedder, and creating a retriever—yet you immediately gain full GraphRAG capabilities. For educational or local experimentation, lightweight tools like nano-graphrag or community repos (e.g., example-graphrag) unpack the same end-to-end pipeline in just a few hundred lines of Python. This system leverages the power of graph databases or knowledge graphs to store and query interconnected data. In GraphRAG, the retrieval phase doesn’t just pull relevant documents or snippets; it analyzes and retrieves nodes and edges from a graph that represents 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 supporting complex queries that involve multiple hops or relationships.

Retrieval system

The retrieval system in GraphRAG 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.

GraphRAG 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 GraphRAG 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 GraphRAG 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 GraphRAG 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 ML 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 ML models can be instrumental in maintaining and updating the knowledge graph efficiently.

Building a knowledge graph can significantly improve complex and multihop retrieval. Typically, this is conducted by extracting semantic triples based on the Resource Description Framework data model. This consists of subject-predicate-object expressions. Foundation models are quite good at extracting these triples, so these types of knowledge graphs can now be constructed at scale. You can see this process visualized in Figure 6-3.

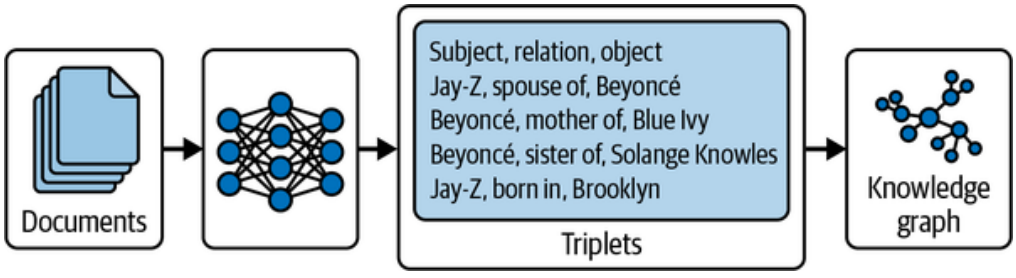

Figure 6-3. Knowledge graph construction workflow. Documents are processed by a model to extract semantic triples in the form of subject-relation-object statements (e.g., "Jay-Z, spouse of, Beyoncé"), which are then structured into a knowledge graph to enable efficient semantic querying and reasoning.

To make it even more approachable, building a basic GraphRAG pipeline today is surprisingly straightforward thanks to open source tooling.

Microsoft's own GraphRAG library, available via `pip install graphrag`, offers a command-line workflow for indexing and querying document collections—no extensive setup required. For instance, after initializing your project and indexing using their CLI, you can run:

```
pip install graphrag
mkdir -p ./ragtest/input
curl https://www.gutenberg.org/ebooks/103.txt.utf-8 -o
 ./ragtest/input/book.txt
graphrag init --root ./ragtest
graphrag index --root ./ragtest

graphrag query \
--root ./ragtest \
--method global \
--query "What are the key themes in this novel?"

graphrag query \
--root ./ragtest \
--method local \
--query "Who is Phileas Fogg and what motivates his jou
```

This instantly gives you global insights and local context over your texts—without writing a single line of Python. If you prefer more control, the Neo4j GraphRAG Python package lets you set up a full GraphRAG pipeline in code. With a few lines (connecting to Neo4j, defining an embedder and retriever, then querying), you get powerful graph-enhanced RAG capabilities. For developers interested in lightweight or educational implementations, there are smaller community projects like nano-graphrag and example repos (e.g., example-graphrag) that unpack the core pipeline in a few hundred lines of Python.

While this is great for experimentation, many teams want to move from a prototype to a hardened, scalable system. That's where Neo4j shines: it's the most trusted, enterprise-grade graph database available. Its native graph storage and index-free adjacency architecture ensures near-constant traversal performance—even as the graph scales to billions of nodes and relationships. Production deployments often use Neo4j Enterprise or AuraDB, offering clustering, fault-tolerance, ACID (atomicity, consistency, isolation, and durability) compliance, and multiregion support. Once you've used the Neo4j

GraphRAG Python tooling or Cypher-based setup to extract entities and define relationships, there's a smooth path to a scalable deployment:

- Populate at scale via Cypher: use `CREATE` and `MERGE` statements to build clean, deduplicated graphs.
- Incremental loading logic ensures you can update with new data without duplication.
- Scale performance through Neo4j's read/write clustering, cache sharding, and optimized query planner.

In short, Neo4j makes transitioning from notebook prototypes to production-grade graph-backed RAG pipelines straightforward—without sacrificing performance, reliability, or maintainability.

Once you've defined your ontology and extracted entities and relationships, it's time to populate your knowledge graph. In Neo4j, this is done using the Cypher `CREATE` clause, which lets you specify nodes with labels and properties and then link them via directed relationships. Best practice is to first load or match existing nodes—ensuring you don't duplicate entities—and then issue separate `CREATE` statements for each relationship, as shown in the following example. By organizing your script into discrete steps (create nodes → match nodes → create relationships), you maintain clarity and can more easily debug or extend your graph as it grows:

```
// Create nodes for concepts and entities
CREATE (:Concept {name: 'Artificial Intelligence'});
CREATE (:Concept {name: 'Machine Learning'});
CREATE (:Concept {name: 'Deep Learning'});
CREATE (:Concept {name: 'Neural Networks'});
CREATE (:Concept {name: 'Computer Vision'});
CREATE (:Concept {name: 'Natural Language Processing'})

CREATE (:Tool {name: 'TensorFlow', creator: 'Google'});
CREATE (:Tool {name: 'PyTorch', creator: 'Facebook'});
CREATE (:Model {name: 'BERT', year: 2018});
CREATE (:Model {name: 'ResNet', year: 2015});

// Create relationships between concepts
MATCH
  (ai:Concept {name:'Artificial Intelligence'}),
  (ml:Concept {name:'Machine Learning'})
CREATE (ml)-[:SUBSET_OF]->(ai);
```

```
MATCH
  (ml:Concept {name:'Machine Learning'}),
  (dl:Concept {name:'Deep Learning'})
CREATE (dl)-[:SUBSET_OF]->(ml);

MATCH
  (dl:Concept {name:'Deep Learning'}),
  (nn:Concept {name:'Neural Networks'})
CREATE (nn)-[:USED_IN]->(dl);

MATCH
  (ai:Concept {name:'Artificial Intelligence'}),
  (cv:Concept {name:'Computer Vision'})
CREATE (cv)-[:APPLICATION_OF]->(ai);

MATCH
  (ai:Concept {name:'Artificial Intelligence'}),
  (nlp:Concept {name:'Natural Language Processing'})
CREATE (nlp)-[:APPLICATION_OF]->(ai);

// Create relationships to tools and models
MATCH
  (tensorflow:Tool {name:'TensorFlow'}),
  (nn:Concept {name:'Neural Networks'})
CREATE (tensorflow)-[:IMPLEMENTS]->(nn);

MATCH
  (pytorch:Tool {name:'PyTorch'}),
  (nn:Concept {name:'Neural Networks'})
CREATE (pytorch)-[:IMPLEMENTS]->(nn);

MATCH
  (nlp:Concept {name:'Natural Language Processing'}),
  (bert:Model {name:'BERT'})
CREATE (bert)-[:BELONGS_TO]->(nlp);

MATCH
  (cv:Concept {name:'Computer Vision'}),
  (resnet:Model {name:'ResNet'})
CREATE (resnet)-[:BELONGS_TO]->(cv);

MATCH
  (tensorflow:Tool {name:'TensorFlow'}),
  (bert:Model {name:'BERT'})
CREATE (bert)-[:BUILT_WITH]->(tensorflow);
```

```
MATCH
  (pytorch:Tool {name:'PyTorch'}),
  (resnet:Model {name:'ResNet'})
CREATE (resnet)-[:BUILT_WITH]->(pytorch);

// Query for finding relationships between concepts
MATCH path = shortestPath(
  (concept1:Concept {name: 'Natural Language Processing
      {name: 'Deep Learning'})
)
RETURN path;

// Query for finding all models that use TensorFlow
MATCH (model:Model)-[:BUILT_WITH]->(tool:Tool {name: 'T
RETURN model.name AS model, model.year AS year;
```

Once loaded, your knowledge graph supports multihop traversals (e.g., `shortestPath` queries) and rich relationship patterns that far exceed what a flat table or vector store can express. This foundation enables advanced GraphRAG workflows—where an agent can traverse the graph at runtime to gather context spanning several degrees of separation—unlocking truly powerful reasoning over structured knowledge. 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. As you can see in Figure 6-4, when answering complex queries, the controller can traverse the graph and perform multihop reasoning over structured data, expanding the range and complexity of questions these types of systems can answer.

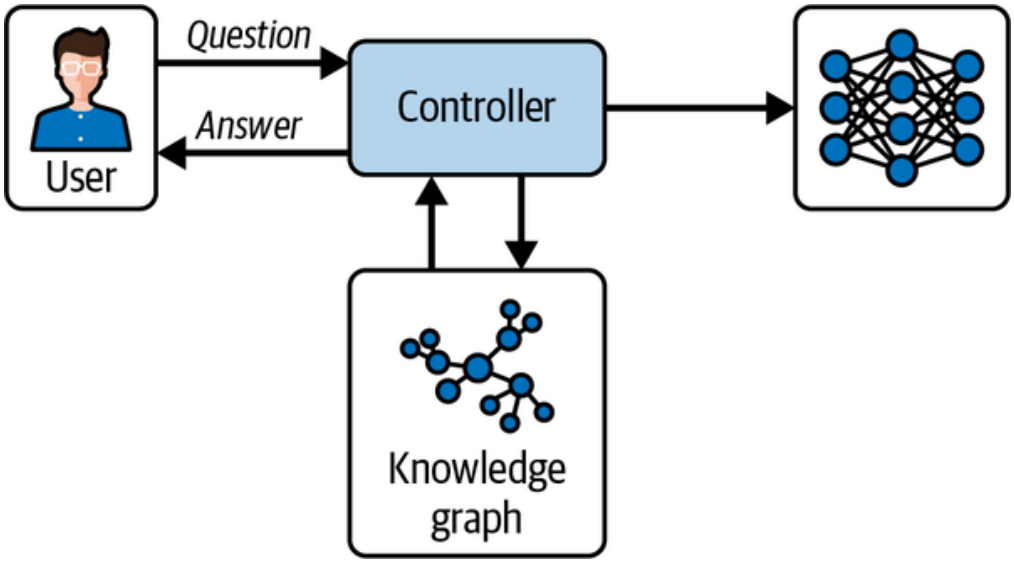

Figure 6-4. Answering questions with knowledge graphs. The user's question is routed through a controller that queries the knowledge graph for relevant structured information, combines it with the language model's reasoning capabilities, and returns a contextually rich, multihop informed answer.

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 are a significant step forward 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 GraphRAG systems. However, the dynamic nature of these graphs also introduces specific challenges that need careful consideration. This section explores the potential benefits and risks associated with dynamic knowledge graphs.

As the developer, it is important to apply careful consideration to choose the most appropriate design to retrieve the appropriate context for handling incoming tasks efficiently. A knowledge graph is reasonably easy to prototype, but getting one ready for production is a significant undertaking.

Recent advances in model architectures are pushing context windows to unprecedented lengths, allowing LLMs to "remember" and process entire documents in a single pass. For example, Google's Gemini 2.5 and OpenAI's GPT-4.1 now support up to *one million* tokens—roughly 750,000 words or over 2,500 pages—enabling retrieval-free generation of very large contexts. Similarly, index-free RAG systems embed their own retrieval logic into long-context models such as GPT-4.1, effectively performing chunking and

relevance scoring internally without external vector stores or inverted indices. Embedding knowledge directly into these extended contexts can simplify pipelines: rather than orchestrating separate retrieval and ranking nodes, an agent can load entire knowledge bases (e.g., policy manuals or technical specs) directly into the prompt and rely on the model's attention mechanisms to surface relevant passages.

However, these retrieval-free approaches come with trade-offs. Processing millions of tokens in one shot demands substantial compute and can introduce latency and cost challenges—sometimes negating the simplicity gains of removing external retrieval. In addition, there is no guarantee that a given model will correctly identify the one piece of relevant information from such a large context window. Do not be surprised if larger models, larger context windows, and more compute make elaborate text search and semantic search over vector databases obsolete, but in the meantime, the community consensus remains that hybrid architectures retain value: even with multipage context windows, RAG can outperform pure long-context models on fact-seeking queries and enterprise use cases, especially when memory freshness or precision ranking is critical. In practice, many production systems combine extended context windows with selective retrieval nodes—leveraging the best of both worlds to balance performance, cost, and factual accuracy.

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 operated effectively and reasoned over, can 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 GraphRAG 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.

Note-Taking

With this technique, the foundation model is prompted to specifically inject notes on the input context without trying to answer the question.[1] 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 Figure 6-5, in a traditional, "vanilla" approach, the model is provided with the context and a question, and it produces an answer. With 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 6-5 illustrates how note-taking enhances standard inference workflows by interleaving model-generated notes alongside the context before producing a final answer.

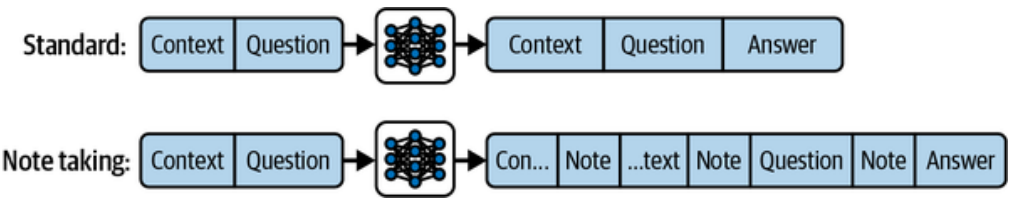

Figure 6-5. Note-taking workflows. In the standard approach, the model processes context and question together to produce an answer directly. In the note-taking approach, the model first generates notes summarizing or elaborating on parts of the context and the question, and then produces the final answer —enabling deeper reasoning and improved task performance.

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 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 Jack Lanchantin et al., "Learning to Reason and Memorize with Self-Notes", arXiv, May 1, 2023.