**Chapter 14\_ Knowledge Retrieval (RAG)**

Chapter 14: Knowledge Retrieval (RAG)

LLMs exhibit substantial capabilities in generating human-like text. However, their knowledge base is typically confined to the data on which they were trained, limiting their access to real-time information, specific company data, or highly specialized details. Knowledge Retrieval (RAG, or Retrieval Augmented Generation), addresses this limitation. RAG enables LLMs to access and integrate external, current, and context-specific information, thereby enhancing the accuracy, relevance, and factual basis of their outputs.

For AI agents, this is crucial as it allows them to ground their actions and responses in real-time, verifiable data beyond their static training. This capability enables them to perform complex tasks accurately, such as accessing the latest company policies to answer a specific question or checking current inventory before placing an order. By integrating external knowledge, RAG transforms agents from simple conversationalists into effective, data-driven tools capable of executing meaningful work.

**Knowledge Retrieval (RAG) Pattern Overview**

The Knowledge Retrieval (RAG) pattern significantly enhances the capabilities of LLMs by granting them access to external knowledge bases before generating a response. Instead of relying solely on their internal, pre-trained knowledge, RAG allows LLMs to "look up" information, much like a human might consult a book or search the internet. This process empowers LLMs to provide more accurate, up-to-date, and verifiable answers.

When a user poses a question or gives a prompt to an AI system using RAG, the query isn't sent directly to the LLM. Instead, the system first scours a vast external knowledge base—a highly organized library of documents, databases, or web pages—for relevant information. This search is not a simple keyword match; it's a "semantic search" that understands the user's intent and the meaning behind their words. This initial search pulls out the most pertinent snippets or "chunks" of information. These extracted pieces are then "augmented," or added, to the original prompt, creating a richer, more informed query. Finally, this enhanced prompt is sent to the LLM. With this additional context, the LLM can generate a response that is not only fluent and natural but also factually grounded in the retrieved data.

The RAG framework provides several significant benefits. It allows LLMs to access up-to-date information, thereby overcoming the constraints of their static training data. This approach also reduces the risk of "hallucination"—the generation of false information—by grounding responses in verifiable data. Moreover, LLMs can utilize specialized knowledge found in internal company documents or wikis. A vital advantage of this process is the capability to offer "citations," which pinpoint the exact source of information, thereby enhancing the trustworthiness and verifiability of the AI's responses..

To fully appreciate how RAG functions, it's essential to understand a few core concepts (see Fig.1):

**Embeddings**: In the context of LLMs, embeddings are numerical representations of text, such as words, phrases, or entire documents. These representations are in the form of a vector, which is a list of numbers. The key idea is to capture the semantic meaning and the relationships between different pieces of text in a mathematical space. Words or phrases with similar meanings will have embeddings that are closer to each other in this vector space. For instance, imagine a simple 2D graph. The word "cat" might be represented by the coordinates (2, 3), while "kitten" would be very close at (2.1, 3.1). In contrast, the word "car" would have a distant coordinate like (8, 1), reflecting its different meaning. In reality, these embeddings are in a much higher-dimensional space with hundreds or even thousands of dimensions, allowing for a very nuanced understanding of language.

**Text Similarity:** Text similarity refers to the measure of how alike two pieces of text are. This can be at a surface level, looking at the overlap of words (lexical similarity), or at a deeper, meaning-based level. In the context of RAG, text similarity is crucial for finding the most relevant information in the knowledge base that corresponds to a user's query. For instance, consider the sentences: "What is the capital of France?" and "Which city is the capital of France?". While the wording is different, they are asking the same question. A good text similarity model would recognize this and assign a high similarity score to these two sentences, even though they only share a few words. This is often calculated using the embeddings of the texts.

**Semantic Similarity and Distance:** Semantic similarity is a more advanced form of text similarity that focuses purely on the meaning and context of the text, rather than just the words used. It aims to understand if two pieces of text convey the same concept or idea. Semantic distance is the inverse of this; a high semantic similarity implies a low semantic distance, and vice versa. In RAG, semantic search relies on finding documents with the smallest semantic distance to the user's query. For instance, the phrases "a furry feline companion" and "a domestic cat" have no words in common besides "a". However, a model that understands semantic similarity would recognize that they refer to the same thing and would consider them to be highly similar. This is because their embeddings would be very close in the vector space, indicating a small semantic distance. This is the "smart search" that allows RAG to find relevant information even when the user's wording doesn't exactly match the text in the knowledge base.

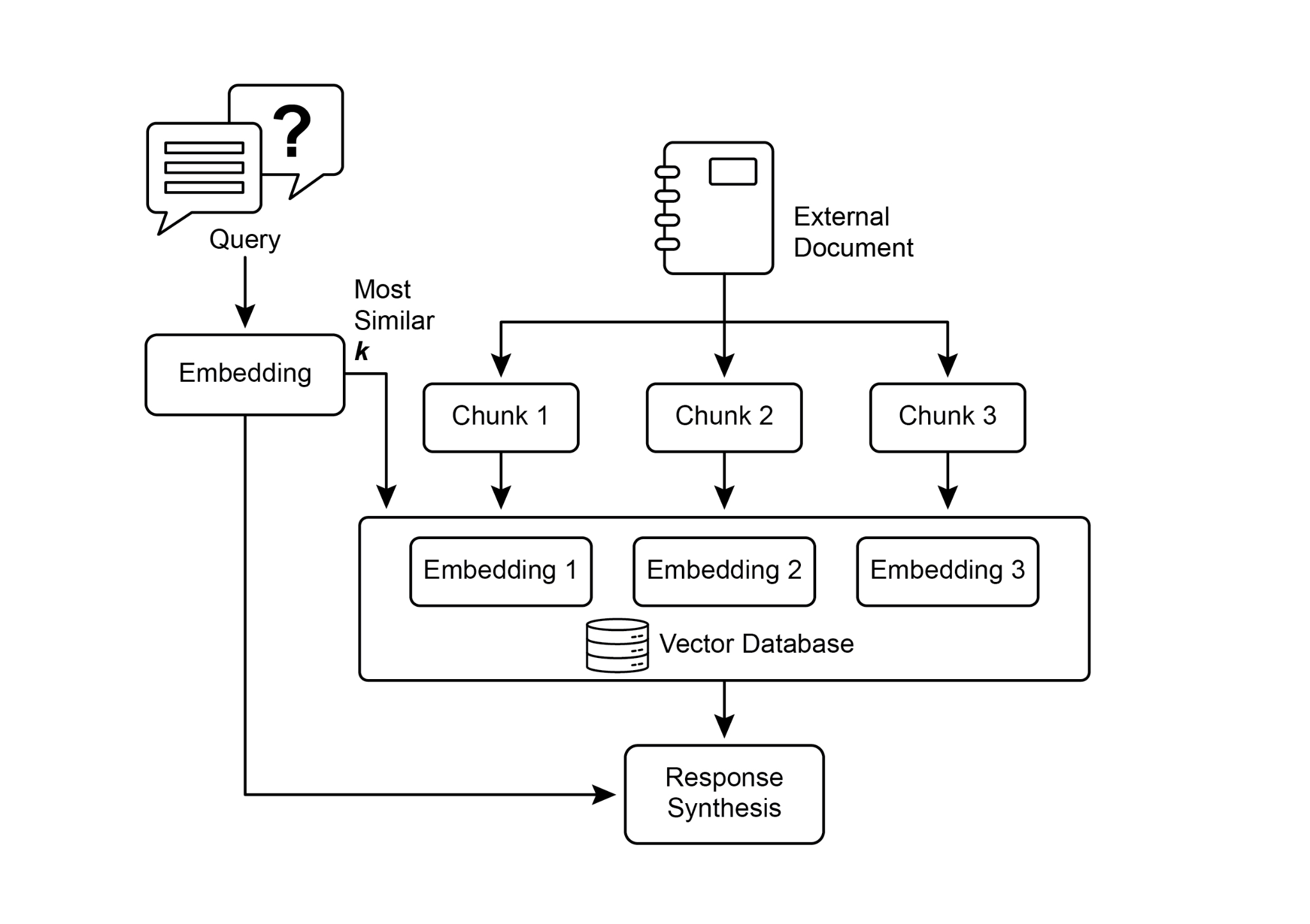


Fig.1: RAG Core Concepts: Chunking, Embeddings, and Vector Database

**Chunking of Documents:** Chunking is the process of breaking down large documents into smaller, more manageable pieces, or "chunks." For a RAG system to work efficiently, it cannot feed entire large documents into the LLM. Instead, it processes these smaller chunks. The way documents are chunked is important for preserving the context and meaning of the information. For instance, instead of treating a 50-page user manual as a single block of text, a chunking strategy might break it down into sections, paragraphs, or even sentences. For instance, a section on "Troubleshooting" would be a separate chunk from the "Installation Guide." When a user asks a question about a specific problem, the RAG system can then retrieve the most relevant troubleshooting chunk, rather than the entire manual. This makes the retrieval process faster and the information provided to the LLM more focused and relevant to the user's immediate need. Once documents are chunked, the RAG system must employ a retrieval technique to find the most relevant pieces for a given query. The primary method is vector search, which uses embeddings and semantic distance to find chunks that are conceptually similar to the user's question. An older, but still valuable, technique is BM25, a keyword-based algorithm that ranks chunks based on term frequency without understanding semantic meaning. To get the best of both worlds, hybrid search approaches are often used, combining the keyword precision of BM25 with the contextual understanding of semantic search. This fusion allows for more robust and accurate retrieval, capturing both literal matches and conceptual relevance.

**Vector databases:** A vector database is a specialized type of database designed to store and query embeddings efficiently. After documents are chunked and converted into embeddings, these high-dimensional vectors are stored in a vector database. Traditional retrieval techniques, like keyword-based search, are excellent at finding documents containing exact words from a query but lack a deep understanding of language. They wouldn't recognize that "furry feline companion" means "cat." This is where vector databases excel. They are built specifically for semantic search. By storing text as numerical vectors, they can find results based on conceptual meaning, not just keyword overlap. When a user's query is also converted into a vector, the database uses highly optimized algorithms (like HNSW - Hierarchical Navigable Small World) to rapidly search through millions of vectors and find the ones that are "closest" in meaning. This approach is far superior for RAG because it uncovers relevant context even if the user's phrasing is completely different from the source documents. In essence, while other techniques search for words, vector databases search for meaning. This technology is implemented in various forms, from managed databases like Pinecone and Weaviate to open-source solutions such as Chroma DB, Milvus, and Qdrant. Even existing databases can be augmented with vector search capabilities, as seen with Redis, Elasticsearch, and Postgres (using the pgvector extension). The core retrieval mechanisms are often powered by libraries like Meta AI's FAISS or Google Research's ScaNN, which are fundamental to the efficiency of these systems.

**RAG's Challenges:** Despite its power, the RAG pattern is not without its challenges. A primary issue arises when the information needed to answer a query is not confined to a single chunk but is spread across multiple parts of a document or even several documents. In such cases, the retriever might fail to gather all the necessary context, leading to an incomplete or inaccurate answer. The system's effectiveness is also highly dependent on the quality of the chunking and retrieval process; if irrelevant chunks are retrieved, it can introduce noise and confuse the LLM. Furthermore, effectively synthesizing information from potentially contradictory sources remains a significant hurdle for these systems. Besides that, another challenge is that RAG requires the entire knowledge base to be pre-processed and stored in specialized databases, such as vector or graph databases, which is a considerable undertaking. Consequently, this knowledge requires periodic reconciliation to remain up-to-date, a crucial task when dealing with evolving sources like company wikis. This entire process can have a noticeable impact on performance, increasing latency, operational costs, and the number of tokens used in the final prompt.

In summary, the Retrieval-Augmented Generation (RAG) pattern represents a significant leap forward in making AI more knowledgeable and reliable. By seamlessly integrating an external knowledge retrieval step into the generation process, RAG addresses some of the core limitations of standalone LLMs. The foundational concepts of embeddings and semantic similarity, combined with retrieval techniques like keyword and hybrid search, allow the system to intelligently find relevant information, which is made manageable through strategic chunking. This entire retrieval process is powered by specialized vector databases designed to store and efficiently query millions of embeddings at scale. While challenges in retrieving fragmented or contradictory information persist, RAG empowers LLMs to produce answers that are not only contextually appropriate but also anchored in verifiable facts, fostering greater trust and utility in AI.

**Graph RAG:** GraphRAG is an advanced form of Retrieval-Augmented Generation that utilizes a knowledge graph instead of a simple vector database for information retrieval. It answers complex queries by navigating the explicit relationships (edges) between data entities (nodes) within this structured knowledge base. A key advantage is its ability to synthesize answers from information fragmented across multiple documents, a common failing of traditional RAG. By understanding these connections, GraphRAG provides more contextually accurate and nuanced responses.

Use cases include complex financial analysis, connecting companies to market events, and scientific research for discovering relationships between genes and diseases. The primary drawback, however, is the significant complexity, cost, and expertise required to build and maintain a high-quality knowledge graph. This setup is also less flexible and can introduce higher latency compared to simpler vector search systems. The system's effectiveness is entirely dependent on the quality and completeness of the underlying graph structure. Consequently, GraphRAG offers superior contextual reasoning for intricate questions but at a much higher implementation and maintenance cost. In summary, it excels where deep, interconnected insights are more critical than the speed and simplicity of standard RAG.

**Agentic RAG:** An evolution of this pattern, known as **Agentic RAG** (see Fig.2), introduces a reasoning and decision-making layer to significantly enhance the reliability of information extraction. Instead of just retrieving and augmenting, an "agent"—a specialized AI component—acts as a critical gatekeeper and refiner of knowledge. Rather than passively accepting the initially retrieved data, this agent actively interrogates its quality, relevance, and completeness, as illustrated by the following scenarios.

First, an agent excels at reflection and source validation. If a user asks, "What is our company's policy on remote work?" a standard RAG might pull up a 2020 blog post alongside the official 2025 policy document. The agent, however, would analyze the documents' metadata, recognize the 2025 policy as the most current and authoritative source, and discard the outdated blog post before sending the correct context to the LLM for a precise answer.

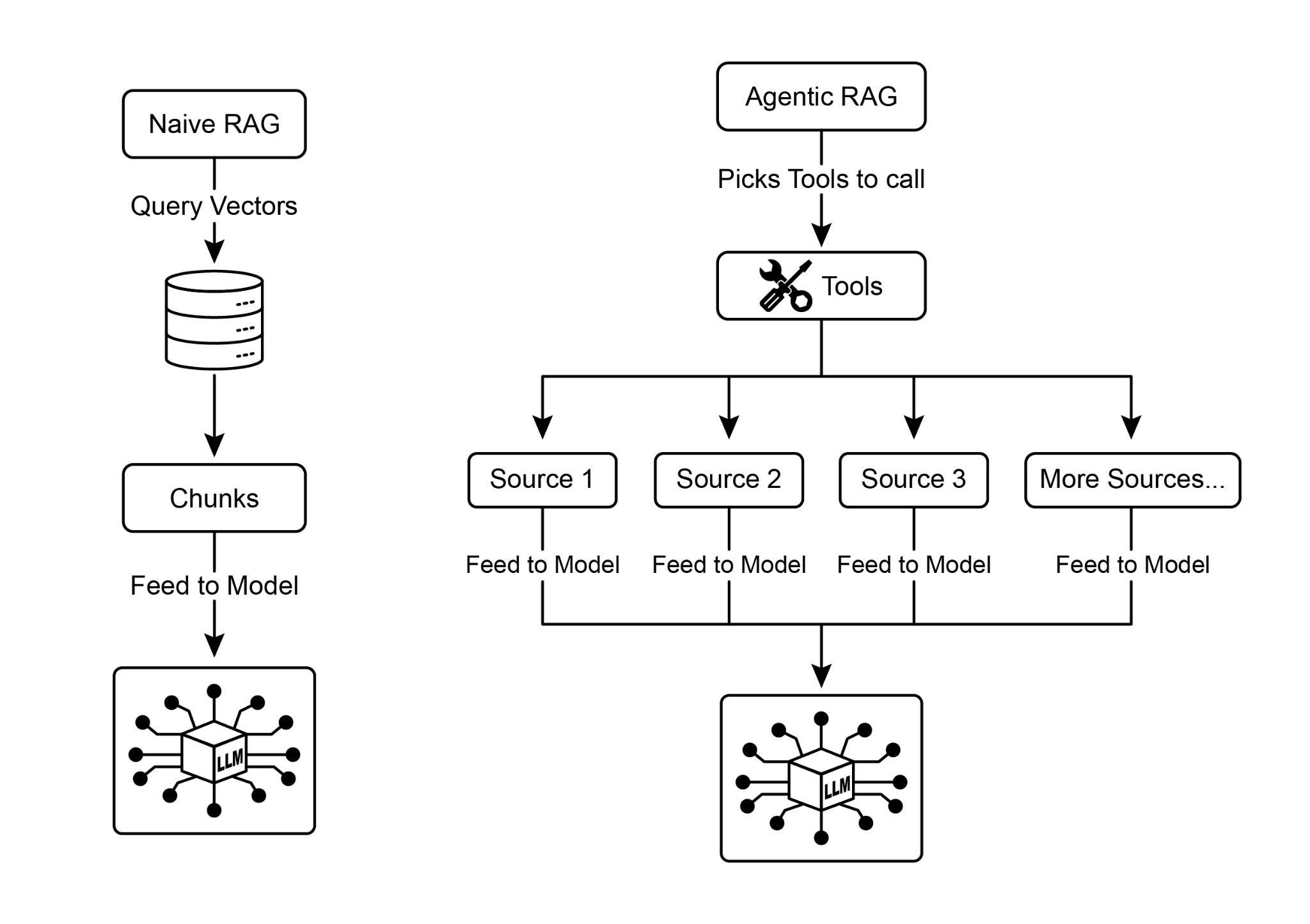


Fig.2: Agentic RAG introduces a reasoning agent that actively evaluates, reconciles, and refines retrieved information to ensure a more accurate and trustworthy final response.

Second, an agent is adept at reconciling knowledge conflicts. Imagine a financial analyst asks, "What was Project Alpha's Q1 budget?" The system retrieves two documents: an initial proposal stating a €50,000 budget and a finalized financial report listing it as €65,000. An Agentic RAG would identify this contradiction, prioritize the financial report as the more reliable source, and provide the LLM with the verified figure, ensuring the final answer is based on the most accurate data.

Third, an agent can perform multi-step reasoning to synthesize complex answers. If a user asks, "How do our product's features and pricing compare to Competitor X's?" the agent would decompose this into separate sub-queries. It would initiate distinct searches for its own product's features, its pricing, Competitor X's features, and Competitor X's pricing. After gathering these individual pieces of information, the agent would synthesize them into a structured, comparative context before feeding it to the LLM, enabling a comprehensive response that a simple retrieval could not have produced.

Fourth, an agent can identify knowledge gaps and use external tools. Suppose a user asks, "What was the market's immediate reaction to our new product launched yesterday?" The agent searches the internal knowledge base, which is updated weekly, and finds no relevant information. Recognizing this gap, it can then activate a tool—such as a live web-search API—to find recent news articles and social media sentiment. The agent then uses this freshly gathered external information to provide an up-to-the-minute answer, overcoming the limitations of its static internal database.

**Challenges of Agentic RAG:** While powerful, the agentic layer introduces its own set of challenges. The primary drawback is a significant increase in complexity and cost. Designing, implementing, and maintaining the agent's decision-making logic and tool integrations requires substantial engineering effort and adds to computational expenses. This complexity can also lead to increased latency, as the agent's cycles of reflection, tool use, and multi-step reasoning take more time than a standard, direct retrieval process. Furthermore, the agent itself can become a new source of error; a flawed reasoning process could cause it to get stuck in useless loops, misinterpret a task, or improperly discard relevant information, ultimately degrading the quality of the final response.

**In summary: Agentic RAG represents a sophisticated evolution of the standard retrieval pattern, transforming it from a passive data pipeline into an active, problem-solving framework. By embedding a reasoning layer that can evaluate sources, reconcile conflicts, decompose complex questions, and use external tools, agents dramatically improve the reliability and depth of the generated answers. This advancement makes the AI more trustworthy and capable, though it comes with important trade-offs in system complexity, latency, and cost that must be carefully managed.**

**Practical Applications & Use Cases**

Knowledge Retrieval (RAG) is changing how Large Language Models (LLMs) are utilized across various industries, enhancing their ability to provide more accurate and contextually relevant responses.

Applications include:

* **Enterprise Search and Q&A:** Organizations can develop internal chatbots that respond to employee inquiries using internal documentation such as HR policies, technical manuals, and product specifications. The RAG system extracts relevant sections from these documents to inform the LLM's response.
* **Customer Support and Helpdesks:** RAG-based systems can offer precise and consistent responses to customer queries by accessing information from product manuals, frequently asked questions (FAQs), and support tickets. This can reduce the need for direct human intervention for routine issues.
* **Personalized Content Recommendation:** Instead of basic keyword matching, RAG can identify and retrieve content (articles, products) that is semantically related to a user's preferences or previous interactions, leading to more relevant recommendations.
* **News and Current Events Summarization:** LLMs can be integrated with real-time news feeds. When prompted about a current event, the RAG system retrieves recent articles, allowing the LLM to produce an up-to-date summary.

By incorporating external knowledge, RAG extends the capabilities of LLMs beyond simple communication to function as knowledge processing systems.

**Hands-On Code Example (ADK)**

To illustrate the Knowledge Retrieval (RAG) pattern, let's see three examples.

First, is how to use Google Search to do RAG and ground LLMs to search results. Since RAG involves accessing external information, the Google Search tool is a direct example of a built-in retrieval mechanism that can augment an LLM's knowledge.

|  |
| --- |
| from google.adk.tools import google\_search  from google.adk.agents import Agent  search\_agent = Agent(  name="research\_assistant",  model="gemini-2.0-flash-exp",  instruction="You help users research topics. When asked, use the Google Search tool",  tools=[google\_search]  ) |

Second, this section explains how to utilize Vertex AI RAG capabilities within the Google ADK. The code provided demonstrates the initialization of VertexAiRagMemoryService from the ADK. This allows for establishing a connection to a Google Cloud Vertex AI RAG Corpus. The service is configured by specifying the corpus resource name and optional parameters such as SIMILARITY\_TOP\_K and VECTOR\_DISTANCE\_THRESHOLD. These parameters influence the retrieval process. SIMILARITY\_TOP\_K defines the number of top similar results to be retrieved. VECTOR\_DISTANCE\_THRESHOLD sets a limit on the semantic distance for the retrieved results. This setup enables agents to perform scalable and persistent semantic knowledge retrieval from the designated RAG Corpus. The process effectively integrates Google Cloud's RAG functionalities into an ADK agent, thereby supporting the development of responses grounded in factual data.

|  |
| --- |
| # Import the necessary VertexAiRagMemoryService class from the google.adk.memory module.  from google.adk.memory import VertexAiRagMemoryService  RAG\_CORPUS\_RESOURCE\_NAME = "projects/your-gcp-project-id/locations/us-central1/ragCorpora/your-corpus-id"  # Define an optional parameter for the number of top similar results to retrieve.  # This controls how many relevant document chunks the RAG service will return.  SIMILARITY\_TOP\_K = 5  # Define an optional parameter for the vector distance threshold.  # This threshold determines the maximum semantic distance allowed for retrieved results;  # results with a distance greater than this value might be filtered out.  VECTOR\_DISTANCE\_THRESHOLD = 0.7  # Initialize an instance of VertexAiRagMemoryService.  # This sets up the connection to your Vertex AI RAG Corpus.  # - rag\_corpus: Specifies the unique identifier for your RAG Corpus.  # - similarity\_top\_k: Sets the maximum number of similar results to fetch.  # - vector\_distance\_threshold: Defines the similarity threshold for filtering results.  memory\_service = VertexAiRagMemoryService(  rag\_corpus=RAG\_CORPUS\_RESOURCE\_NAME,  similarity\_top\_k=SIMILARITY\_TOP\_K,  vector\_distance\_threshold=VECTOR\_DISTANCE\_THRESHOLD  ) |

**Hands-On Code Example (LangChain)**

Third, let's walk through a complete example using LangChain.

|  |
| --- |
| import os  import requests  from typing import List, Dict, Any, TypedDict  from langchain\_community.document\_loaders import TextLoader  from langchain\_core.documents import Document  from langchain\_core.prompts import ChatPromptTemplate  from langchain\_core.output\_parsers import StrOutputParser  from langchain\_community.embeddings import OpenAIEmbeddings  from langchain\_community.vectorstores import Weaviate  from langchain\_openai import ChatOpenAI  from langchain.text\_splitter import CharacterTextSplitter  from langchain.schema.runnable import RunnablePassthrough  from langgraph.graph import StateGraph, END  import weaviate  from weaviate.embedded import EmbeddedOptions  import dotenv  # Load environment variables (e.g., OPENAI\_API\_KEY)  dotenv.load\_dotenv()  # Set your OpenAI API key (ensure it's loaded from .env or set here)  # os.environ["OPENAI\_API\_KEY"] = "YOUR\_OPENAI\_API\_KEY"  # --- 1. Data Preparation (Preprocessing) ---  # Load data  url = "https://github.com/langchain-ai/langchain/blob/master/docs/docs/how\_to/state\_of\_the\_union.txt"  res = requests.get(url)  with open("state\_of\_the\_union.txt", "w") as f:  f.write(res.text)  loader = TextLoader('./state\_of\_the\_union.txt')  documents = loader.load()  # Chunk documents  text\_splitter = CharacterTextSplitter(chunk\_size=500, chunk\_overlap=50)  chunks = text\_splitter.split\_documents(documents)  # Embed and store chunks in Weaviate  client = weaviate.Client(  embedded\_options = EmbeddedOptions()  )  vectorstore = Weaviate.from\_documents(  client = client,  documents = chunks,  embedding = OpenAIEmbeddings(),  by\_text = False  )  # Define the retriever  retriever = vectorstore.as\_retriever()  # Initialize LLM  llm = ChatOpenAI(model\_name="gpt-3.5-turbo", temperature=0)  # --- 2. Define the State for LangGraph ---  class RAGGraphState(TypedDict):  question: str  documents: List[Document]  generation: str  # --- 3. Define the Nodes (Functions) ---  def retrieve\_documents\_node(state: RAGGraphState) -> RAGGraphState:  """Retrieves documents based on the user's question."""  question = state["question"]  documents = retriever.invoke(question)  return {"documents": documents, "question": question, "generation": ""}  def generate\_response\_node(state: RAGGraphState) -> RAGGraphState:  """Generates a response using the LLM based on retrieved documents."""  question = state["question"]  documents = state["documents"]  # Prompt template from the PDF  template = """You are an assistant for question-answering tasks.  Use the following pieces of retrieved context to answer the question.  If you don't know the answer, just say that you don't know.  Use three sentences maximum and keep the answer concise.  Question: {question}  Context: {context}  Answer:  """  prompt = ChatPromptTemplate.from\_template(template)  # Format the context from the documents  context = "\n\n".join([doc.page\_content for doc in documents])  # Create the RAG chain  rag\_chain = prompt | llm | StrOutputParser()  # Invoke the chain  generation = rag\_chain.invoke({"context": context, "question": question})  return {"question": question, "documents": documents, "generation": generation}  # --- 4. Build the LangGraph Graph ---  workflow = StateGraph(RAGGraphState)  # Add nodes  workflow.add\_node("retrieve", retrieve\_documents\_node)  workflow.add\_node("generate", generate\_response\_node)  # Set the entry point  workflow.set\_entry\_point("retrieve")  # Add edges (transitions)  workflow.add\_edge("retrieve", "generate")  workflow.add\_edge("generate", END)  # Compile the graph  app = workflow.compile()  # --- 5. Run the RAG Application ---  if \_\_name\_\_ == "\_\_main\_\_":  print("\n--- Running RAG Query ---")  query = "What did the president say about Justice Breyer"  inputs = {"question": query}  for s in app.stream(inputs):  print(s)  print("\n--- Running another RAG Query ---")  query\_2 = "What did the president say about the economy?"  inputs\_2 = {"question": query\_2}  for s in app.stream(inputs\_2):  print(s) |

This Python code illustrates a Retrieval-Augmented Generation (RAG) pipeline implemented with LangChain and LangGraph. The process begins with the creation of a knowledge base derived from a text document, which is segmented into chunks and transformed into embeddings. These embeddings are then stored in a Weaviate vector store, facilitating efficient information retrieval. A StateGraph in LangGraph is utilized to manage the workflow between two key functions: `retrieve\_documents\_node` and `generate\_response\_node`. The `retrieve\_documents\_node` function queries the vector store to identify relevant document chunks based on the user's input. Subsequently, the `generate\_response\_node` function utilizes the retrieved information and a predefined prompt template to produce a response using an OpenAI Large Language Model (LLM). The `app.stream` method allows the execution of queries through the RAG pipeline, demonstrating the system's capacity to generate contextually relevant outputs.

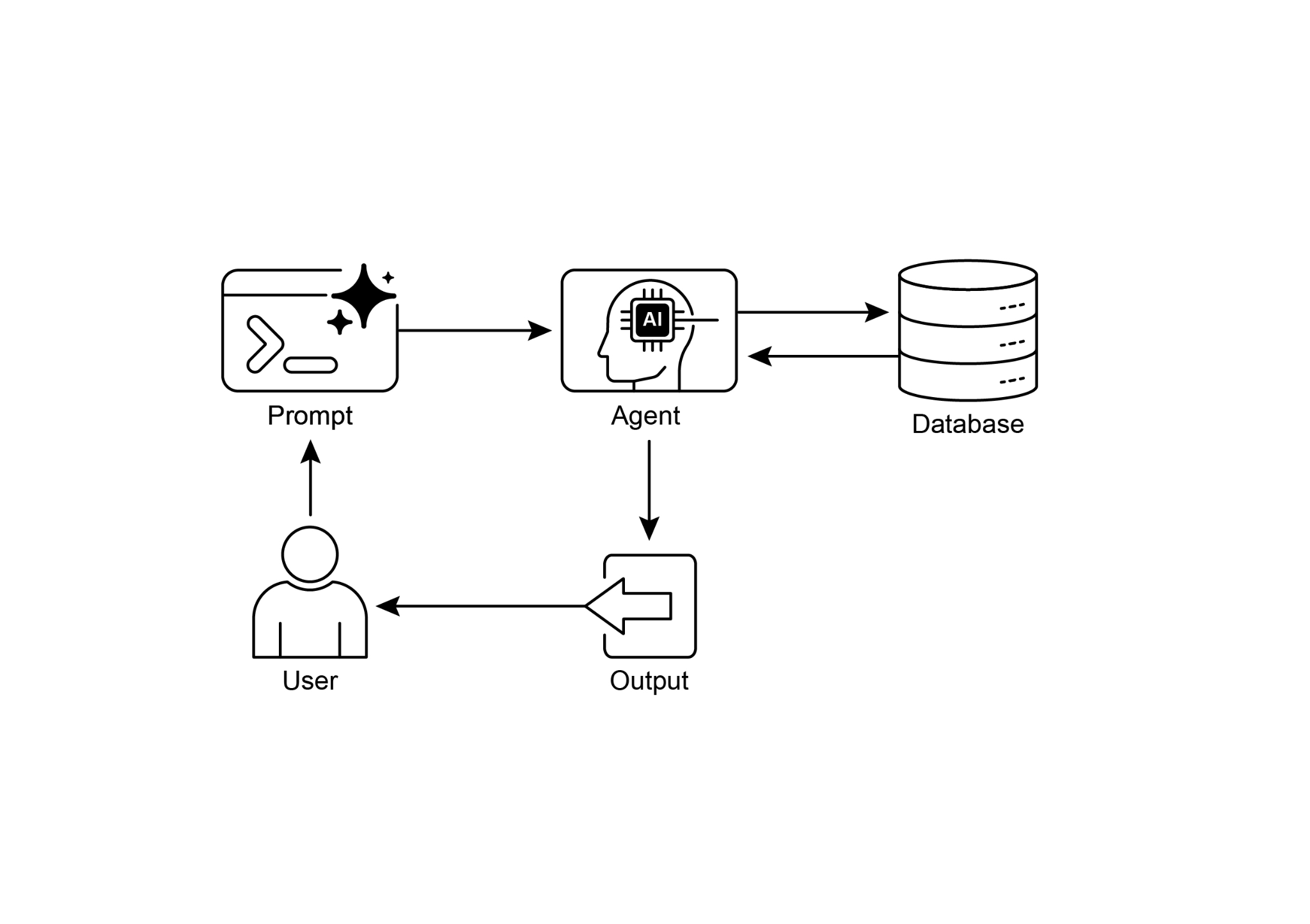
**At Glance**

**What:** LLMs possess impressive text generation abilities but are fundamentally limited by their training data. This knowledge is static, meaning it doesn't include real-time information or private, domain-specific data. Consequently, their responses can be outdated, inaccurate, or lack the specific context required for specialized tasks. This gap restricts their reliability for applications demanding current and factual answers.

**Why:** The Retrieval-Augmented Generation (RAG) pattern provides a standardized solution by connecting LLMs to external knowledge sources. When a query is received, the system first retrieves relevant information snippets from a specified knowledge base. These snippets are then appended to the original prompt, enriching it with timely and specific context. This augmented prompt is then sent to the LLM, enabling it to generate a response that is accurate, verifiable, and grounded in external data. This process effectively transforms the LLM from a closed-book reasoner into an open-book one, significantly enhancing its utility and trustworthiness.

**Rule of thumb:** Use this pattern when you need an LLM to answer questions or generate content based on specific, up-to-date, or proprietary information that was not part of its original training data. It is ideal for building Q&A systems over internal documents, customer support bots, and applications requiring verifiable, fact-based responses with citations.

**Visual summary**



Knowledge Retrieval pattern: an AI agent to query and retrieve information from structured databases

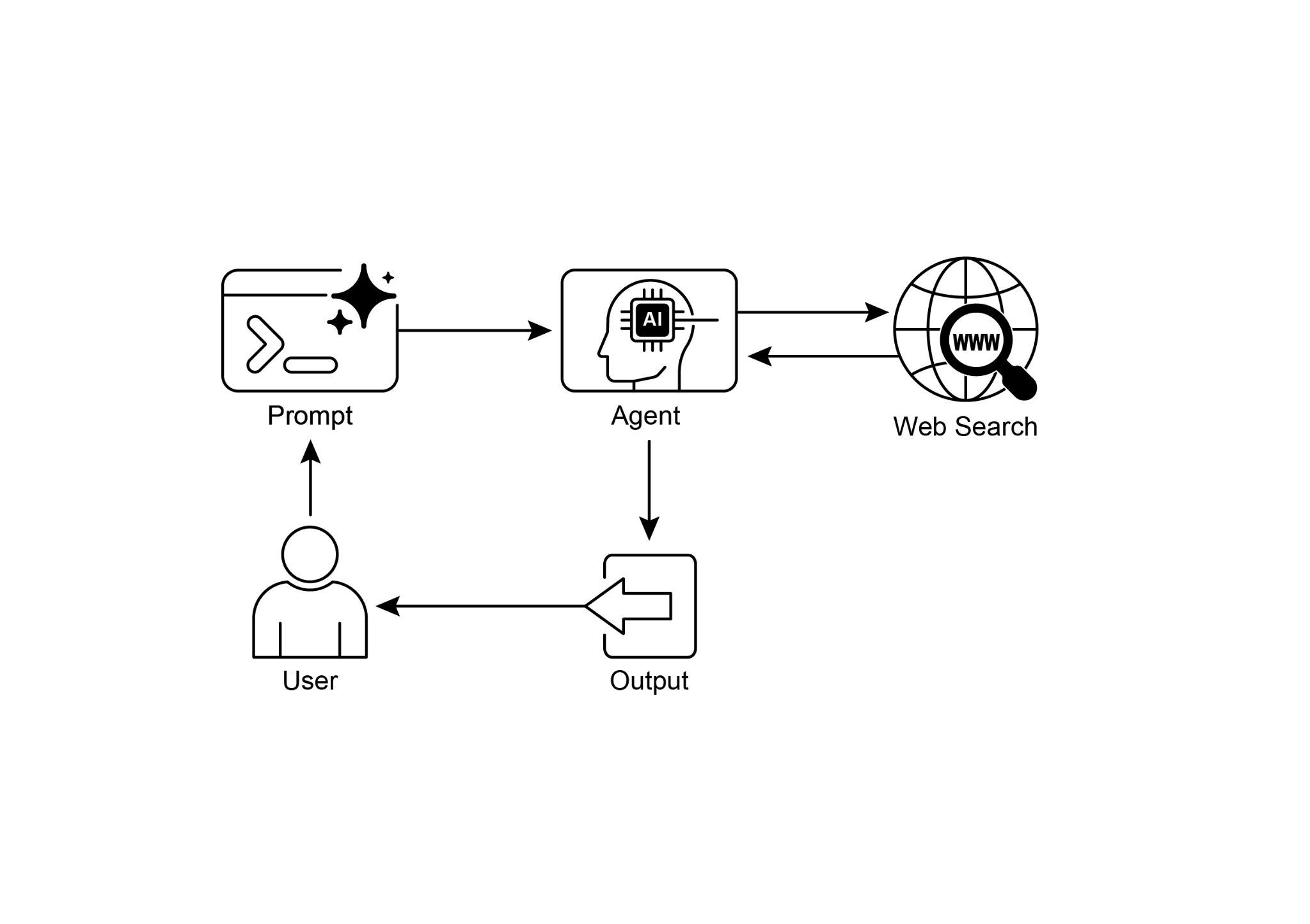


Fig. 3: Knowledge Retrieval pattern: an AI agent to find and synthesize information from the public internet in response to user queries.

**Key Takeaways**

* Knowledge Retrieval (RAG) enhances LLMs by allowing them to access external, up-to-date, and specific information.
* The process involves Retrieval (searching a knowledge base for relevant snippets) and Augmentation (adding these snippets to the LLM's prompt).
* RAG helps LLMs overcome limitations like outdated training data, reduces "hallucinations," and enables domain-specific knowledge integration.
* RAG allows for attributable answers, as the LLM's response is grounded in retrieved sources.
* GraphRAG leverages a knowledge graph to understand the relationships between different pieces of information, allowing it to answer complex questions that require synthesizing data from multiple sources.
* Agentic RAG moves beyond simple information retrieval by using an intelligent agent to actively reason about, validate, and refine external knowledge, ensuring a more accurate and reliable answer.
* Practical applications span enterprise search, customer support, legal research, and personalized recommendations.

**Conclusion**

In conclusion, Retrieval-Augmented Generation (RAG) addresses the core limitation of a Large Language Model's static knowledge by connecting it to external, up-to-date data sources. The process works by first retrieving relevant information snippets and then augmenting the user's prompt, enabling the LLM to generate more accurate and contextually aware responses. This is made possible by foundational technologies like embeddings, semantic search, and vector databases, which find information based on meaning rather than just keywords. By grounding outputs in verifiable data, RAG significantly reduces factual errors and allows for the use of proprietary information, enhancing trust through citations.

An advanced evolution, Agentic RAG, introduces a reasoning layer that actively validates, reconciles, and synthesizes retrieved knowledge for even greater reliability. Similarly, specialized approaches like GraphRAG leverage knowledge graphs to navigate explicit data relationships, allowing the system to synthesize answers to highly complex, interconnected queries. This agent can resolve conflicting information, perform multi-step queries, and use external tools to find missing data. While these advanced methods add complexity and latency, they drastically improve the depth and trustworthiness of the final response. Practical applications for these patterns are already transforming industries, from enterprise search and customer support to personalized content delivery. Despite the challenges, RAG is a crucial pattern for making AI more knowledgeable, reliable, and useful. Ultimately, it transforms LLMs from closed-book conversationalists into powerful, open-book reasoning tools.

**References**

1. Lewis, P., et al. (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. <https://arxiv.org/abs/2005.11401>
2. Google AI for Developers Documentation. *Retrieval Augmented Generation -* [*https://cloud.google.com/vertex-ai/generative-ai/docs/rag-engine/rag-overview*](https://cloud.google.com/vertex-ai/generative-ai/docs/rag-engine/rag-overview)
3. Retrieval-Augmented Generation with Graphs (GraphRAG), <https://arxiv.org/abs/2501.00309>
4. LangChain and LangGraph: Leonie Monigatti, "Retrieval-Augmented Generation (RAG): From Theory to LangChain Implementation," [*https://medium.com/data-science/retrieval-augmented-generation-rag-from-theory-to-langchain-implementation-4e9bd5f6a4f2*](https://medium.com/data-science/retrieval-augmented-generation-rag-from-theory-to-langchain-implementation-4e9bd5f6a4f2)
5. Google Cloud Vertex AI RAG Corpus [*https://cloud.google.com/vertex-ai/generative-ai/docs/rag-engine/manage-your-rag-corpus#corpus-management*](https://cloud.google.com/vertex-ai/generative-ai/docs/rag-engine/manage-your-rag-corpus#corpus-management)

**第14章\_知识检索（RAG）**

第14章：知识检索（RAG）

大语言模型（LLMs）在生成类人文本方面展现出强大的能力。然而，它们的知识库通常局限于训练数据，这限制了它们获取实时信息、特定公司数据或高度专业化细节的能力。知识检索（RAG，即检索增强生成）解决了这一局限性。RAG使大语言模型能够访问和整合外部、最新且特定于上下文的信息，从而提高其输出的准确性、相关性和事实依据。

对于AI智能体来说，这至关重要，因为它使它们能够将自己的行动和响应建立在实时、可验证的数据之上，而不仅仅局限于静态训练。这种能力使它们能够准确地执行复杂任务，例如访问最新的公司政策以回答特定问题，或在下单前查看当前库存。通过整合外部知识，检索增强生成（RAG）将智能体从简单的对话者转变为能够执行有意义工作的高效、数据驱动的工具。

**知识检索（RAG）模式概述**

知识检索（RAG）模式通过让大语言模型（LLMs）在生成响应之前访问外部知识库，显著增强了它们的能力。RAG 允许大语言模型“查找”信息，而不是仅仅依赖其内部的预训练知识，这就像人类查阅书籍或搜索互联网一样。这个过程使大语言模型能够提供更准确、最新且可验证的答案。

当用户使用RAG向AI系统提出问题或给出提示时，查询不会直接发送到大型语言模型（LLM）。相反，系统首先会在庞大的外部知识库（一个高度组织化的文档、数据库或网页库）中搜索相关信息。这种搜索并非简单的关键词匹配； 这是一种“语义搜索”，它能理解用户的意图和其话语背后的含义。这种初始搜索会提取出最相关的信息片段或“块”。然后，这些提取出来的内容会被“增强”，即添加到原始提示中，从而形成一个更丰富、更有信息依据的查询。最后，这个增强后的提示会被发送给大语言模型（LLM）。有了这些额外的上下文，大语言模型就能生成不仅流畅自然，而且在事实层面有检索数据支撑的回应。

检索增强生成（RAG）框架具有诸多显著优势。它使大语言模型（LLMs）能够获取最新信息，从而克服其静态训练数据的局限性。这种方法还通过将回答建立在可验证的数据基础上，降低了“幻觉”（即生成虚假信息）的风险。此外，大语言模型可以利用公司内部文件或维基百科中包含的专业知识。这一过程的一个重要优势是能够提供“引用”，这些引用可以pin定信息的确切来源，从而提高AI回答的可信度和可验证性。

要充分理解检索增强生成（RAG）的工作原理，必须了解几个核心概念（见图1）：

**嵌入**：在大语言模型（LLM）的语境中，嵌入是文本（如单词、短语或整个文档）的数值表示。这些表示采用向量的形式，即一个数字列表。其核心思想是在数学空间中捕捉不同文本片段之间的语义含义和关系。意思相近的单词或短语在这个向量空间中的嵌入会彼此更接近。例如，想象一个简单的二维图。单词“cat”（猫）可能由坐标(2, 3)表示，而“kitten”（小猫）则非常接近，为(2.1, 3.1)。相比之下，单词“car”（汽车）的坐标会很远，如(8, 1)，反映出其不同的含义。实际上，这些嵌入存在于一个更高维的空间中，有数百甚至数千个维度，从而能够对语言进行非常细致的理解。

**文本相似度：**文本相似度指的是衡量两段文本相似程度的指标。这可以是表面层面的，即查看词汇的重叠情况（词汇相似度），也可以是更深层次的、基于语义的层面。在检索增强生成（RAG）的语境中，文本相似度对于在知识库中找到与用户查询相对应的最相关信息至关重要。例如，考虑以下句子：“法国的首都是什么？”和“哪个城市是法国的首都？”虽然表述不同，但它们问的是同一个问题。一个好的文本相似度模型会识别出这一点，并为这两个句子赋予较高的相似度得分，即使它们只有几个词相同。这通常是通过文本的嵌入向量来计算的。

**语义相似度和距离：**语义相似度是一种更高级的文本相似度形式，它纯粹关注文本的含义和语境，而不仅仅是所使用的词汇。它旨在理解两段文本是否传达相同的概念或想法。语义距离则是其相反的概念； 高语义相似度意味着低语义距离，反之亦然。在检索增强生成（RAG）中，语义搜索依赖于找到与用户查询语义距离最小的文档。例如，短语“毛茸茸的猫科伙伴”和“家猫”除了“a”之外没有共同的单词。然而，理解语义相似度的模型会认识到它们指的是同一事物，并会认为它们高度相似。这是因为它们的嵌入在向量空间中会非常接近，表明语义距离很小。这就是使RAG即使在用户的措辞与知识库中的文本不完全匹配时也能找到相关信息的“智能搜索”。

图1：RAG核心概念：分块、嵌入和向量数据库

**文档分块：**分块是将大型文档分解成更小、更易于管理的部分或“块”的过程。为了使检索增强生成（RAG）系统高效运行，它不能将整个大型文档直接输入到大语言模型（LLM）中，而是处理这些较小的块。文档的分块方式对于保留信息的上下文和含义非常重要。例如，分块策略可能不会将50页的用户手册视为一个单一的文本块，而是将其分解为章节、段落甚至句子。例如，“故障排除”部分将与“安装指南”分开作为一个独立的块。当用户询问特定问题时，RAG系统可以检索最相关的故障排除块，而不是整个手册。这使得检索过程更快，并且提供给LLM的信息更加聚焦，与用户的即时需求更相关。文档分块后，RAG系统必须采用一种检索技术来为给定查询找到最相关的部分。主要方法是向量搜索，它使用嵌入和语义距离来查找在概念上与用户问题相似的块。一种较旧但仍然有价值的技术是BM25，这是一种基于关键词的算法，它在不理解语义含义的情况下根据词频对块进行排名。为了兼顾两者的优势，通常会使用混合搜索方法，将BM25的关键词精度与语义搜索的上下文理解相结合。这种融合允许进行更强大、更准确的检索，既捕捉字面匹配，又捕捉概念相关性。

**向量数据库：**向量数据库是一种专门设计的数据库类型，旨在高效存储和查询嵌入向量。在文档被分块并转换为嵌入向量后，这些高维向量会被存储在向量数据库中。传统的检索技术，如基于关键词的搜索，在查找包含查询中确切词汇的文档方面表现出色，但缺乏对语言的深入理解。它们无法识别“毛茸茸的猫科伙伴”就是“猫”的意思。而向量数据库在这方面表现卓越。它们专门为语义搜索而构建。通过将文本存储为数值向量，它们可以基于概念含义而非仅仅是关键词重叠来查找结果。当用户的查询也被转换为向量时，数据库会使用高度优化的算法（如分层可通航小世界图 - HNSW）快速搜索数百万个向量，并找到含义“最接近”的向量。这种方法在检索增强生成（RAG）中具有明显优势，因为即使用户的表述与源文档完全不同，它也能揭示相关上下文。本质上，其他技术搜索的是词汇，而向量数据库搜索的是含义。这项技术以各种形式实现，从像Pinecone和Weaviate这样的托管数据库，到如Chroma DB、Milvus和Qdrant等开源解决方案。甚至现有的数据库也可以通过向量搜索功能进行增强，如Redis、Elasticsearch和Postgres（使用pgvector扩展）。核心检索机制通常由Meta AI的FAISS或Google Research的ScaNN等库提供支持，这些库是这些系统高效运行的基础。

**检索增强生成（RAG）面临的挑战：**尽管RAG模式功能强大，但并非没有挑战。当回答查询所需的信息并非局限于单个文本块，而是分散在文档的多个部分甚至多个文档中时，就会出现一个主要问题。在这种情况下，检索器可能无法收集到所有必要的上下文信息，从而导致答案不完整或不准确。系统的有效性也高度依赖于分块和检索过程的质量； 如果检索到不相关的片段，可能会引入噪声并使大语言模型（LLM）产生混淆。此外，有效地整合来自潜在矛盾来源的信息，仍然是这些系统面临的重大障碍。除此之外，另一个挑战是，检索增强生成（RAG）要求对整个知识库进行预处理，并存储在专门的数据库中，如向量数据库或图数据库，这是一项相当艰巨的任务。因此，这些知识需要定期进行协调，以保持最新状态，这在处理像公司维基这样不断发展的信息源时是一项至关重要的任务。整个过程可能会对性能产生显著影响，增加延迟、运营成本以及最终提示中使用的令牌数量。

总之，检索增强生成（RAG）模式代表了使AI更具知识和可靠性方面的重大飞跃。通过将外部知识检索步骤无缝集成到生成过程中，RAG解决了独立大语言模型（LLM）的一些核心局限性。嵌入和语义相似性的基本概念，再结合关键词和混合搜索等检索技术，使系统能够智能地找到相关信息，并通过策略性分块使其易于管理。整个检索过程由专门的向量数据库驱动，这些数据库旨在大规模存储和高效查询数百万个嵌入。尽管在检索碎片化或相互矛盾的信息方面仍然存在挑战，但RAG使大语言模型能够生成不仅在上下文中合适，而且基于可验证事实的答案，从而增强了对AI的信任和实用性。

**图检索增强生成（Graph RAG）：**图检索增强生成（GraphRAG）是检索增强生成（RAG）的一种高级形式，它利用知识图谱而非简单的向量数据库进行信息检索。它通过在这个结构化知识库中导航数据实体（节点）之间的显式关系（边）来回答复杂查询。一个关键优势是它能够从分散在多个文档中的信息中综合生成答案，而这正是传统RAG常见的短板。通过理解这些关联，图检索增强生成（GraphRAG）能够提供更具上下文准确性和细微差别的响应。

用例包括复杂的金融分析、将公司与市场事件关联起来，以及用于发现基因与疾病之间关系的科学研究。然而，主要缺点是构建和维护高质量知识图谱所需的显著复杂性、成本和专业知识。与更简单的向量搜索系统相比，这种设置的灵活性也较低，并且可能会引入更高的延迟。系统的有效性完全取决于底层图结构的质量和完整性。因此，GraphRAG在处理复杂问题时提供了卓越的上下文推理能力，但实施和维护成本要高得多。总之，在深度、相互关联的见解比标准RAG的速度和简单性更重要的情况下，它表现出色。

**智能体RAG：**这种模式的一种演进，即**智能体RAG**（见图2），引入了推理和决策层，以显著提高信息提取的可靠性。与单纯的检索和增强不同，“智能体”（一种专门的AI组件）充当知识的关键把关者和精炼者。该智能体并非被动接受最初检索到的数据，而是积极审视其质量、相关性和完整性，以下场景对此进行了说明。

首先，智能体擅长反思和来源验证。如果用户询问“我们公司的远程工作政策是什么？”，标准的检索增强生成（RAG）可能会同时调出2020年的博客文章和2025年的官方政策文件。然而，智能体将分析文档的元数据，识别出2025年的政策是最新且最具权威性的来源，并在将正确的上下文发送给大语言模型（LLM）以获取准确答案之前，舍弃过时的博客文章。

图2：代理式RAG引入了一个推理代理，该代理会主动评估、协调和提炼检索到的信息，以确保最终响应更加准确和可信。

其次，智能体善于调和知识冲突。想象一下，一位金融分析师问道：“阿尔法项目的Q1预算是多少？”系统检索到两份文件：一份初始提案显示预算为50,000欧元，一份最终财务报告则将其列为65,000欧元。智能体增强检索生成（Agentic RAG）会识别出这一矛盾，将财务报告列为更可靠的来源，并向大语言模型（LLM）提供经过核实的数据，确保最终答案基于最准确的数据。

第三，智能体可以进行多步推理，以合成复杂的答案。如果用户询问：“我们产品的功能和定价与竞争对手X相比如何？”代理会将此分解为单独的子查询。它会分别对自身产品的特性、定价、竞争对手X的特性和竞争对手X的定价展开独立搜索。在收集到这些单独的信息片段后，代理会将它们综合成一个结构化的比较语境，然后再将其提供给大语言模型（LLM），从而实现简单检索无法产生的全面回应。

第四，智能体可以识别知识缺口并使用外部工具。假设用户询问：“市场对我们昨天推出的新产品有何即时反应？”智能体搜索每周更新的内部知识库，未找到相关信息。发现这一信息缺口后，它可以激活一个工具，如实时网络搜索 API，来查找最新的新闻文章和社交媒体情绪。然后，智能体利用这些新收集的外部信息提供最新的答案，克服其静态内部数据库的局限性。

**自主式RAG的挑战：**虽然自主层功能强大，但它也带来了自身的一系列挑战。主要缺点是复杂性和成本显著增加。设计、实施和维护代理的决策逻辑及工具集成需要大量的工程工作，并增加了计算成本。这种复杂性还可能导致延迟增加，因为代理的反思、工具使用和多步推理周期比标准的直接检索过程耗时更长。此外，代理本身可能成为新的错误来源； 有缺陷的推理过程可能导致它陷入无用的循环、误解任务或不恰当地丢弃相关信息，最终降低最终回复的质量。

**总结如下：**代理式RAG代表了标准检索模式的复杂演进，将其从被动的数据管道转变为主动的问题解决框架。通过嵌入一个能够评估来源、协调冲突、分解复杂问题并使用外部工具的推理层，代理显著提高了生成答案的可靠性和深度。这一进步使AI更值得信赖且更有能力，尽管它在系统复杂性、延迟和成本方面带来了重要的权衡，必须谨慎管理。

**实际应用与用例**

知识检索增强生成（RAG）正在改变大语言模型（LLMs）在各行业的应用方式，提升其提供更准确、上下文相关响应的能力。

应用包括：

* **企业搜索与问答：**组织可以开发内部聊天机器人，利用内部留档（如HR政策、技术手册和产品规格）来回应员工的咨询。RAG系统从这些文档中提取相关部分，为大语言模型的回答提供信息。
* **客户支持和帮助台：**基于检索增强生成（RAG）的系统可以通过访问产品手册、常见问题解答（FAQs）和支持工单中的信息，为客户咨询提供准确且一致的响应。这可以减少日常问题对直接人工干预的需求。
* **个性化内容推荐：**与基本的关键词匹配不同，RAG可以识别和检索与用户偏好或之前的互动在语义上相关的内容（文章、产品），从而实现更相关的推荐。
* **新闻与时事摘要：**大语言模型（LLMs）可以与实时新闻源集成。当被问及当前事件时，检索增强生成（RAG）系统会检索最近的文章，使大语言模型能够生成最新的摘要。

通过整合外部知识，检索增强生成（RAG）将大语言模型（LLMs）的能力从简单的通信扩展到作为知识处理系统发挥作用。

**实践代码示例（ADK）**

为了说明知识检索（RAG）模式，让我们来看三个例子。

首先，是如何使用谷歌搜索来进行检索增强生成（RAG），并将大语言模型（LLMs）与搜索结果关联起来。由于RAG涉及访问外部信息，谷歌搜索工具是一个内置检索机制的直接示例，可增强大语言模型的知识。

|  |
| --- |
| from google.adk.tools import google\_search  from google.adk.agents import Agent  search\_agent = Agent(  name="research\_assistant",  model="gemini-2.0-flash-exp",  instruction="你帮助用户研究主题。被询问时，使用谷歌搜索工具",  工具=[谷歌搜索]  ) |

其次，本节将解释如何在 Google ADK 中利用 Vertex AI RAG 功能。所提供的代码展示了如何从 ADK 初始化 VertexAiRagMemoryService。这使得能够与 Google Cloud Vertex AI RAG 语料库建立连接。该服务通过指定语料库资源名称和可选参数（如 SIMILARITY\_TOP\_K 和 VECTOR\_DISTANCE\_THRESHOLD）进行配置。这些参数会影响检索过程。SIMILARITY\_TOP\_K 定义了要检索的最相似结果的数量。VECTOR\_DISTANCE\_THRESHOLD 为检索结果的语义距离设定了一个限制。这种设置使代理能够从指定的 RAG 语料库中执行可扩展且持久的语义知识检索。该过程有效地将 Google Cloud 的 RAG 功能集成到 ADK 代理中，从而支持基于事实数据的响应开发。

|  |
| --- |
| # 从google.adk.memory模块导入必要的VertexAiRagMemoryService类。  from google.adk.memory import VertexAiRagMemoryService  RAG\_CORPUS\_RESOURCE\_NAME = "projects/your-gcp-project-id/locations/us-central1/ragCorpora/your-corpus-id"  # 定义一个可选参数，用于指定要检索的最相似结果的数量。  # 此设置控制RAG服务将返回多少相关文档块。  相似度前K值 = 5  # 定义向量距离阈值的可选参数。  # 此阈值决定了检索结果允许的最大语义距离；  距离大于此值的结果可能会被过滤掉。  向量距离阈值 = 0.7  # 初始化 VertexAiRagMemoryService 的一个实例。  # 此步骤设置与您的 Vertex AI RAG 语料库的连接。  # - rag\_corpus：指定您的RAG语料库的唯一标识符。  # - similarity\_top\_k：设置要获取的相似结果的最大数量。  # - vector\_distance\_threshold：定义用于过滤结果的相似度阈值。  memory\_service = VertexAiRagMemoryService(  rag\_corpus=RAG\_CORPUS\_RESOURCE\_NAME,  similarity\_top\_k=SIMILARITY\_TOP\_K,  vector\_distance\_threshold=VECTOR\_DISTANCE\_THRESHOLD  ) |

**实践代码示例（LangChain）**

第三，让我们通过一个使用LangChain的完整示例来详细了解一下。

|  |
| --- |
| 导入 os  导入请求库  从 typing 导入 List、Dict、Any、TypedDict  从 langchain\_community.document\_loaders 导入 TextLoader  from langchain\_core.documents import Document  from langchain\_core.prompts import ChatPromptTemplate  从 langchain\_core.output\_parsers 导入 StrOutputParser  from langchain\_community.embeddings import OpenAIEmbeddings  from langchain\_community.vectorstores import Weaviate  from langchain\_openai import ChatOpenAI  从langchain.text\_splitter导入CharacterTextSplitter  从langchain.schema.runnable导入RunnablePassthrough  从langgraph.graph导入StateGraph, END  导入 weaviate  从 weaviate.embedded 导入 EmbeddedOptions  导入 dotenv  # 加载环境变量（例如，OPENAI\_API\_KEY）  dotenv.load\_dotenv()  # 设置你的OpenAI API密钥（确保从.env文件加载或在此处设置）  # os.environ["OPENAI\_API\_KEY"] = "YOUR\_OPENAI\_API\_KEY"  # --- 1. 数据准备（预处理） ---  # 加载数据  url = "https://github.com/langchain-ai/langchain/blob/master/docs/docs/how\_to/state\_of\_the\_union.txt"  res = requests.get(url)  with open("state\_of\_the\_union.txt", "w") as f:  f.write(res.text)  loader = TextLoader('./state\_of\_the\_union.txt')  documents = loader.load()  # 分块文档  text\_splitter = CharacterTextSplitter(chunk\_size=500, chunk\_overlap=50)  chunks = text\_splitter.split\_documents(documents)  # 将分块嵌入并存储在 Weaviate 中  client = weaviate.Client(  embedded\_options = EmbeddedOptions()  )  vectorstore = Weaviate.from\_documents(  client = client,  文档 = 块  embedding = OpenAIEmbeddings(),  by\_text = False  )  # 定义检索器  retriever = vectorstore.as\_retriever()  # 初始化大语言模型  llm = ChatOpenAI(model\_name="gpt-3.5-turbo", temperature=0)  # --- 2. 定义LangGraph的状态 ---  class RAGGraphState(TypedDict):  question: str  documents: List[Document]  generation: str  # --- 3. 定义节点（函数） ---  def retrieve\_documents\_node(state: RAGGraphState) -> RAGGraphState:  根据用户的问题检索文档。  question = state["question"]  documents = retriever.invoke(question)  return {"documents": documents, "question": question, "generation": ""}  def generate\_response\_node(state: RAGGraphState) -> RAGGraphState:  根据检索到的文档，使用大语言模型生成响应。  question = state["question"]  documents = state["documents"]  # 来自PDF的提示模板  模板 = """你是一个用于问答任务的助手。"""  使用以下检索到的上下文片段来回答问题。  如果你不知道答案，就直接说不知道。  最多使用三句话，保持答案简洁。  问题：{question}  上下文：{context}  答案：  """  prompt = ChatPromptTemplate.from\_template(template)  # 格式化文档中的上下文  context = "\n\n".join([doc.page\_content for doc in documents])  # 创建RAG链  rag\_chain = prompt | llm | StrOutputParser()  # 调用链  generation = rag\_chain.invoke({"context": context, "question": question})  return {"question": question, "documents": documents, "generation": generation}  # --- 4. 构建LangGraph图 ---  workflow = StateGraph(RAGGraphState)  # 添加节点  workflow.add\_node("retrieve", retrieve\_documents\_node)  workflow.add\_node("generate", generate\_response\_node)  # 设置切入点  workflow.set\_entry\_point("retrieve")  # 添加边（转换）  workflow.add\_edge("retrieve", "generate")  workflow.add\_edge("generate", END)  # 编译图  app = workflow.compile()  # --- 5. 运行RAG应用程序 ---  if \_\_name\_\_ == "\_\_main\_\_":  print("\n--- 正在运行RAG查询 ---")  查询语句为 "总统对布雷耶大法官说了什么"  inputs = {"question": query}  for s in app.stream(inputs):  print(s)  print("\n--- 正在运行另一个RAG查询 ---")  query\_2 = "总统对经济说了什么？"  inputs\_2 = {"question": query\_2}  for s in app.stream(inputs\_2):  print(s) |

这段Python代码展示了一个使用LangChain和LangGraph实现的检索增强生成（RAG）管道。该过程始于从文本文件创建知识库，将其分割成块并转换为嵌入向量。这些嵌入向量随后存储在Weaviate向量存储中，便于高效的信息检索。LangGraph中的StateGraph用于管理两个关键函数之间的工作流程：`retrieve\_documents\_node`和`generate\_response\_node`。`retrieve\_documents\_node`函数查询向量存储，根据用户输入识别相关的文档块。随后，`generate\_response\_node`函数利用检索到的信息和预定义的提示模板，使用OpenAI大语言模型（LLM）生成响应。`app.stream`方法允许通过RAG管道执行查询，展示了系统生成上下文相关输出的能力。

**概览**

**问题：**大语言模型（LLMs）拥有令人印象深刻的文本生成能力，但从根本上受到其训练数据的限制。这些知识是静态的，这意味着它不包含实时信息或特定领域的私有数据。因此，它们的回答可能过时、不准确，或者缺乏专业任务所需的特定上下文。这种差距限制了它们在需要最新和事实性答案的应用中的可靠性。

**原因：**检索增强生成（RAG）模式通过将大语言模型（LLMs）连接到外部知识源，提供了一种标准化的解决方案。当接收到查询时，系统首先从指定的知识库中检索相关信息片段。然后将这些片段附加到原始提示中，用及时且具体的上下文对其进行丰富。这个增强后的提示随后被发送到大语言模型，使其能够生成准确、可验证且基于外部数据的响应。这一过程有效地将大语言模型从闭卷推理器转变为开卷推理器，显著提高了其实用性和可信度。

**经验法则：**当你需要大语言模型（LLM）基于特定、最新或专有信息（这些信息并非其原始训练数据的一部分）来回答问题或生成内容时，请使用此模式。它非常适合基于内部文档构建问答系统、开发客户支持机器人，以及开发需要可验证、基于事实且带有引用的响应的应用程序。

**可视化总结**

知识检索模式：一种从结构化数据库中查询和检索信息的AI智能体

图3：知识检索模式：一个AI智能体，用于响应用户查询，从公共互联网中查找和整合信息。

**要点总结**

* 知识检索（RAG）通过允许大语言模型（LLMs）访问外部、最新和特定的信息来增强它们的能力。
* 该过程包括检索（在知识库中搜索相关片段）和增强（将这些片段添加到大型语言模型的提示中）。
* 检索增强生成（RAG）有助于大语言模型（LLMs）克服训练数据过时等局限性，减少“幻觉”，并实现特定领域知识的整合。
* 检索增强生成（RAG）能够提供可溯源的答案，因为大语言模型（LLM）的回答基于检索到的来源。
* GraphRAG利用知识图谱来理解不同信息片段之间的关系，使其能够回答需要整合来自多个来源数据的复杂问题。
* 能动式检索增强生成（Agentic RAG）超越了简单的信息检索，它利用智能体主动对外部知识进行推理、验证和提炼，从而确保答案更加准确可靠。
* 实际应用涵盖企业搜索、客户支持、法律研究和个性化推荐。

**结论**

总之，检索增强生成（RAG）通过将大语言模型（LLM）连接到外部最新数据源，解决了其静态知识的核心局限性。该过程首先检索相关信息片段，然后增强用户提示，使大语言模型能够生成更准确、更具上下文感知的响应。这得益于嵌入、语义搜索和向量数据库等基础技术，这些技术基于语义而非仅仅关键词来查找信息。通过将输出建立在可验证的数据基础上，RAG显著减少了事实性错误，并允许使用专有信息，通过引用增强了可信度。

一种先进的演进形式——代理式检索增强生成（Agentic RAG）引入了一个推理层，该层会主动验证、协调和整合检索到的知识，以提高可靠性。同样，像图检索增强生成（GraphRAG）这样的专门方法利用知识图谱来梳理明确的数据关系，使系统能够综合回答高度复杂、相互关联的查询。这种代理可以解决冲突信息，执行多步骤查询，并使用外部工具查找缺失数据。虽然这些先进方法增加了复杂性和延迟，但它们极大地提高了最终响应的深度和可信度。这些模式的实际应用已经在改变各个行业，从企业搜索和客户支持到个性化内容交付。尽管面临挑战，但检索增强生成（RAG）是使AI更具知识、更可靠和更有用的关键模式。最终，它将大语言模型（LLMs）从闭卷对话者转变为强大的开卷推理工具。

**参考文献**

1. 刘易斯，P.等人（2020年）。*知识密集型NLP任务的检索增强生成*。<https://arxiv.org/abs/2005.11401>
2. Google开发者AI文档。*检索增强生成 -* [*https://cloud.google.com/vertex-ai/generative-ai/docs/rag-engine/rag-overview*](https://cloud.google.com/vertex-ai/generative-ai/docs/rag-engine/rag-overview)
3. 基于图的检索增强生成（GraphRAG），<https://arxiv.org/abs/2501.00309>
4. LangChain和LangGraph：Leonie Monigatti，《检索增强生成（RAG）：从理论到LangChain实现》，[*https://medium.com/data-science/retrieval-augmented-generation-rag-from-theory-to-langchain-implementation-4e9bd5f6a4f2*](https://medium.com/data-science/retrieval-augmented-generation-rag-from-theory-to-langchain-implementation-4e9bd5f6a4f2)
5. Google Cloud Vertex AI RAG语料库[*https://cloud.google.com/vertex-ai/generative-ai/docs/rag-engine/manage-your-rag-corpus#corpus-management*](https://cloud.google.com/vertex-ai/generative-ai/docs/rag-engine/manage-your-rag-corpus#corpus-management)