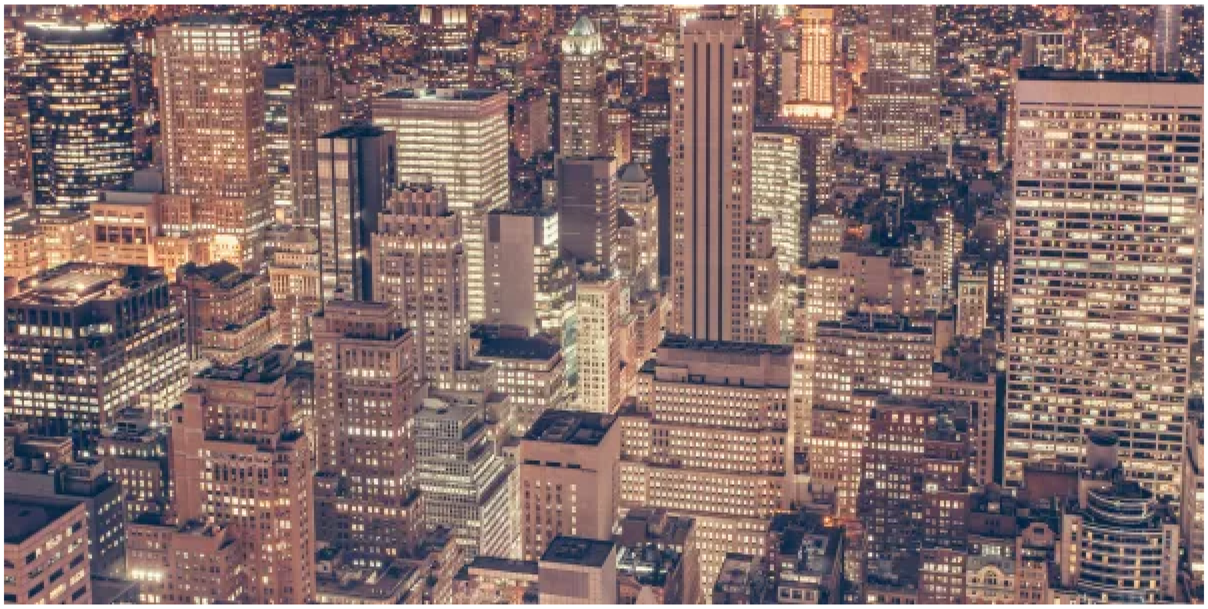
Understanding the Difference Between RAG and AI Agents

Dean Chen · Follow

Published in OLarry · 14 min read · Mar 5, 2025

17


NYC Nighttime

So, everyone's working on enhancing those powerful language models — LLMs — without the need for constant retraining. Two really interesting ways people are tackling this are called **RAG (Retrieval-Augmented Generation)** and **AI Agents**. Now, they might seem a bit alike at first glance, but they're actually quite different. Each has its own strengths and weaknesses, and they're best suited for different kinds of tasks.

Alright, let's get down to brass tacks with a simple example. Picture this: you're hanging out with friends, and someone asks, "Hey, what's the average temperature in New York during summer, and should I pack light or bring a jacket?" You've got a couple of ways to figure that out.

With RAG, it's like having a research assistant. Instead of just relying on what it already knows, it quickly scans through a massive library of historical weather data, picks out the most relevant information, and uses that to give you a precise answer. For example, it might tell you that the average summer temperature in New York during the day is around 85°F (29°C) and pull a list of recommendations from your clothing data. It's not just guessing; it's pulling from reliable, up-to-date sources to make sure you get the most accurate response.

But with an AI Agent, it's like having a super-smart personal travel planner. It doesn't just retrieve information — it actively plans and makes decisions for you. Here's how it works:

1. **Retrieves Historical Data**: It uses RAG to find the average summer temperature in New York.
2. **Checks Real-Time Data**: It pulls the current weather forecast using a weather API.
3. **Analyzes Your Preferences**: It remembers you prefer packing light but hate being cold.

4. **Makes a Recommendation**: Based on the average temperature, current forecast, and your preferences, it suggests, ‘Pack light, but bring a light jacket for cooler evenings.’

Intrigued? Now let’s dive deeper.

What is Retrieval-Augmented Generation (RAG)?

Retrieval-Augmented Generation (RAG) is a technique that enhances the response of LLMs by retrieving relevant information from an external knowledge base before generating an answer. Unlike standard LLMs that rely solely on pre-trained knowledge, RAG provides more up-to-date and domain-specific context to improve accuracy and reliability.

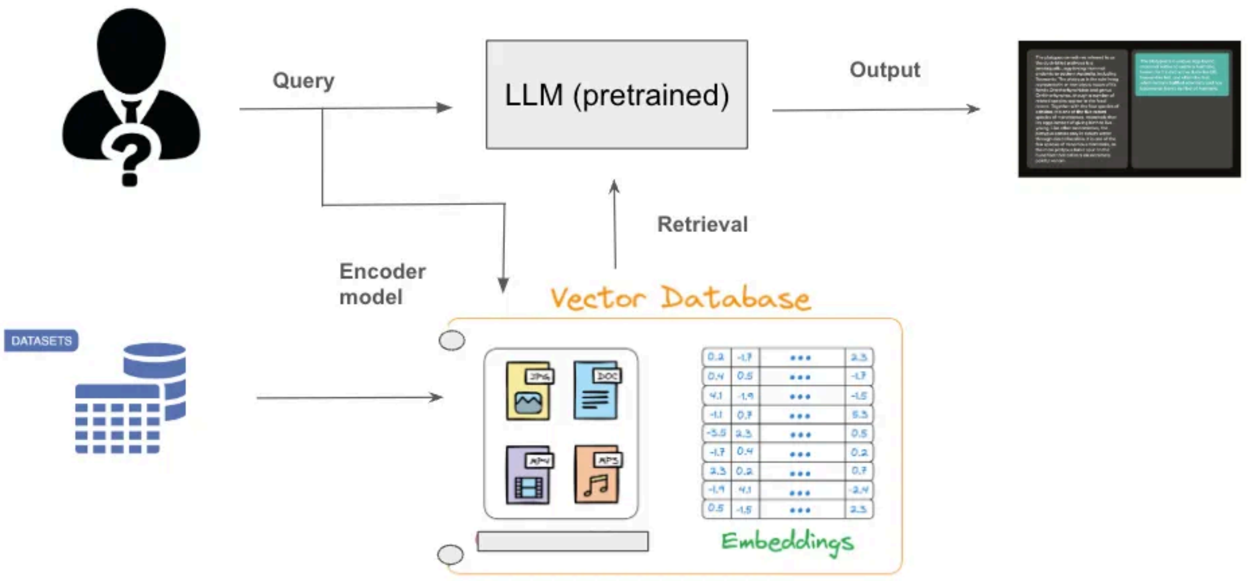
How RAG Works

1. **Async Preprocessing Step:** Before retrieval can happen, the information or documents must first be converted into vector format using an embedding model and stored in a Vector Database (VD). This step happens asynchronously as new documents are ingested into the system.
2. **User Query:** When a user submits a query, the input is also converted into vector format using the same embedding model.
3. **Retrieval Step:** The system searches the Vector Database to find the most semantically similar documents to the input query by leveraging the vector space properties.
4. **Conversion Back to Text:** Once the most relevant documents are retrieved from the Vector Database, they are converted back to text before being fed into the LLM.

5. **Augmentation Step:** The retrieved text data is injected into the LLM's context to enhance its response.
6. **Response Generation:** The LLM processes both the retrieved data and its pre-trained knowledge to generate a more accurate and relevant response.

This method is useful in applications such as customer support, medical research, and financial analysis, where real-time and contextually rich responses are critical.

Here is a diagram illustrating how RAG functions:



- **Data Set:** The knowledge source that the LLM can retrieve information from.
- **Vector Database (VD):** Stores preprocessed documents as vector representations for efficient retrieval.

- **User Query:** The prompt asking for information (e.g., "What phone plan is best?").
- **Embedding Model:** Converts both documents and user queries leveraging Encoder model into vector format for comparison.
- **Retrieval Step:** The system searches the Vector Database for semantically similar documents.
- **Conversion Back to Text:** The retrieved vectors are mapped back to their original textual content.
- **Pre-trained LLM:** The model combines retrieved knowledge with its pre-trained understanding.
- **Final Output:** A response that is more informed and context-aware.

Here is an example of RAG code snippet: (Using LangChain with FAISS for Document Retrieval)

```
from langchain.chat_models import ChatOpenAI
from langchain.vectorstores import FAISS
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.document_loaders import TextLoader
from langchain.chains import RetrievalQA
import os

# Set API Keys
os.environ["OPENAI_API_KEY"] = "your_openai_api_key"

# Load and Embed Documents
documents = TextLoader("knowledge_base.txt").load()
vectorstore = FAISS.from_documents(documents, OpenAIEmbeddings())

# Initialize LLM and Retrieval QA
llm = ChatOpenAI(model="gpt-4", temperature=0)
retriever = vectorstore.as_retriever()
qa_chain = RetrievalQA.from_chain_type(llm=llm, chain_type="stuff", retriever=re
# Ask a Question
question = "What are the key takeaways from the document?"
retrieved_docs = retriever.get_relevant_documents(question)

for i, doc in enumerate(retrieved_docs):
    print(f"Document {i+1}:\n{doc.page_content}\n")

#Augmentation and Response Generation (Pre-trained LLM + Retrieved Context)
response = qa_chain.run(question)
print(response)
```

Limitations of RAG: Context Size Constraints

While RAG is a powerful technique for enhancing LLMs with external knowledge, it's not without its limitations. One of the most significant challenges is its dependency on the LLM's context window size.

- **Context Window Size:** LLMs have a fixed context window (e.g., 4,096 tokens for GPT-3.5 or 8,192 tokens for GPT-4, 128k for GPT-4o, Gemini 1.5 Pro 2 million)*. This limits the amount of information that can be retrieved and processed in a single query.
- **Retrieval vs. Context:** RAG retrieves relevant documents from a knowledge base, but if the combined size of the retrieved documents and the user query exceeds the LLM's context window, the system must truncate or omit information. This can lead to incomplete or suboptimal responses.
- **Workarounds for context size limitation:** Breaking down large documents into smaller, manageable chunks that fit within the context window. Furthermore, using a two-step retrieval process — first retrieving a broad set of documents and then refining the selection to the

most relevant ones. Summarizing or compressing retrieved documents to reduce their size while retaining key information.

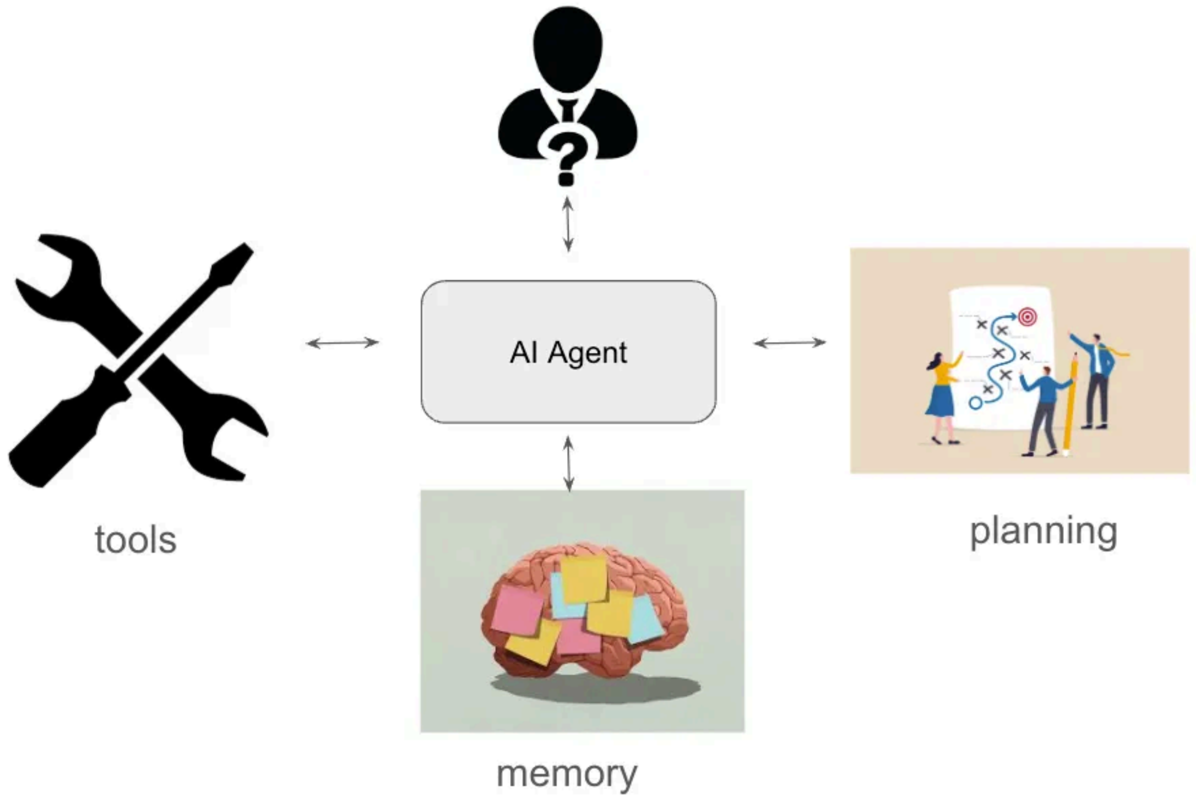
The Power of AI Agents

In contrast, AI Agents has a few key abilities that make it so effective:

1. **Tools —** These could be anything from weather APIs and search engines to even other AI systems, like a RAG system, for pulling specific document-based information. Leverage RAG to research historical weather patterns, then using a real-time API for the current temperature.
2. **Memory —** This allows it to remember past clues and conversations, so it can build context and understand the bigger picture.

- **Short-term Memory:** Stores recent interactions within a single conversation session. This helps maintain context within a limited timeframe.
- **Long-term Memory:** Persists across multiple interactions, enabling AI Agents to recall historical information from previous conversations.
- **Near Real-Time/Streaming Memory:** Enables AI Agents to process and retain information dynamically from continuous streams of input, such as real-time chat logs, sensor data, or event-driven systems.
- **Passive Memory:** Passively stores and retrieves relevant historical information when needed, rather than continuously updating in real time.

3. Planning — AI Agents analyze the problem, figure out the best sequence of actions to solve it, and decide which tools to use and when. It's not about random actions — it's about strategic thinking and careful analysis to get the job done right.



What Kind of Analysis Does an AI Agent Perform?

Some of the analysis performed by the AI Agents:

- **Dependency Analysis:** Determining what tasks rely on each other and sequencing them logically. For example, if you ask an AI Agent to book a flight, it may first check your existing calendar, ensure passport validity, and only then proceed with booking.
- **Constraint Handling:** Taking into account rules, budgets, deadlines, and preferences. If you tell an AI Agent to find the best flight, it can cross-check price limits, layover preferences, and airline loyalty programs.

- **Optimization:** It doesn't just provide options but optimizes them based on multiple factors. A meeting planner AI Agent might suggest a conference room that minimizes travel time between your meetings while ensuring all required attendees can join.
- **Decision Making:** AI Agents can proactively suggest actions, not just provide data. For example, instead of just giving you the weather report, it might say, *"You have an outdoor event tomorrow, and there's a 70% chance of rain. Should I reschedule it or find an indoor venue?"*

AI Agents Go Beyond Information Retrieval — Action

An AI Agent doesn't stop at retrieving information — it can execute actions on your behalf if you allow it. For example:

- **Personal Calendar Planning:** Imagine telling an AI Agent, "I have a gym session tomorrow, a client call in the morning, and I need to book a flight in the afternoon. Plan my day for me." The AI Agent would: Retrieve your existing commitments. Suggest the best slot for your flight booking, avoiding conflicts. Automatically reschedule flexible tasks like gym sessions. And set reminders based on your preferences.
- **Smart Home Automation:** AI Agents can interact with IoT devices to execute actions. For example, *"If the room temperature exceeds 75°F, turn on the air conditioner."*
- **Business Workflow Automation:** AI Agents can automate workflows by integrating with CRM systems, managing customer queries, or handling order processing based on real-time data.

This ability to plan, analyze, and take actions dynamically is what makes AI Agents significantly different from RAG-based retrieval systems.

This concept is also known as **ReAct** (Reasoning and Acting), it is the ability of an AI model to both reason and take actions

Following is an example of AI Agent code snippet: (simulate Customer support use case)

```
from langchain.chat_models import ChatOpenAI
from langchain.agents import Tool, AgentExecutor, create_openai_tools_agent
from langchain.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain.memory import ConversationBufferMemory
import os

# Set API Keys
os.environ["OPENAI_API_KEY"] = "your_openai_api_key"

# Define Tools
def search_knowledge_base(query: str) -> str:
    return f"Here are some articles that might help: [Article 1, Article 2, Arti

def check_order_status(order_id: str) -> str:
    return f"Order {order_id} is out for delivery."

tools = [
    Tool(
        name="SearchKnowledgeBase",
        func=search_knowledge_base,
        description="Useful for finding customer support articles."
    ),
    Tool(
        name="CheckOrderStatus",
        func=check_order_status,
        description="Useful for checking the status of an order."
    )
]

# Initialize LLM and Memory
llm = ChatOpenAI(model="gpt-4", temperature=0)
memory = ConversationBufferMemory(memory_key="chat_history", return_messages=Tru

# Create Agent
prompt = ChatPromptTemplate.from_messages([
    ("system", "You are a helpful customer support AI. Use tools and memory to a
    MessagesPlaceholder(variable_name="chat_history"),
    ("user", "{input}")
])

agent = create_openai_tools_agent(llm, tools, prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, memory=memory, verbose=

# Simulate a Customer Interaction
user_query = "Hi, I need help with my order #12345."
response = agent_executor.invoke({"input": user_query})

print("AI Response:", response["output"])

# Follow-up Query (Demonstrates Memory)
follow_up_query = "Can you also suggest some articles for troubleshooting?"
response = agent_executor.invoke({"input": follow_up_query})

print("AI Response:", response["output"])
```

The Role of Multi-Agent Systems

The field of AI Agents is increasingly focusing on **Multi-Agent Systems (MAS)**, where multiple agents collaborate to achieve complex goals. Multi-Agent Systems leverage the strengths of different AI Agents specialized in specific tasks, allowing for scalability, modularity, and enhanced reasoning.

Several frameworks are driving this multi-agent paradigm, including:

- **LangGraph** — A framework for building multi-agent workflows, enabling agents to interact with structured logic*.
- **CrewAI** — A system designed for orchestrating and managing multiple AI agents working on different tasks*.
- **AWS Bedrock Agents** — A platform offering agent-based AI solutions integrated into AWS infrastructure*.

These frameworks support connected and enterprise-scale AI Agents that can process inputs in parallel or sequential workflows, depending on the complexity of the tasks.

Expanded Frameworks for Multi-Agent Systems

Furthermore, the AI Agent ecosystem is rapidly growing, with several frameworks emerging to support multi-agent collaboration. In addition to LangGraph, CrewAI, and AWS Bedrock Agents, here are a few other notable frameworks:

- **Llama Index:** A framework designed for building and managing multi-agent systems, particularly focused on efficient data retrieval and context management*.
- **Langroid:** A lightweight and flexible framework for orchestrating multi-agent workflows, with a focus on natural language interactions and task delegation*.
- **Google's Multi-Agent Framework:** Recently announced, this framework aims to provide a runtime environment for managing multi-agent systems at scale, including support for SaaS integrations*.

These frameworks are pushing the boundaries of what's possible with multi-agent systems, enabling more complex and scalable AI-driven applications.

Example: Multi-Agent Code Generation with LangGraph

```
from langgraph.graph import Graph
from langchain.agents import Tool
from langchain.chat_models import ChatOpenAI
from langchain.agents import initialize_agent
from langchain.prompts import PromptTemplate
```

```
# Define the CodeGeneratorAgent
def code_generator(query: str) -> str:
    llm = ChatOpenAI(model="gpt-3.5-turbo")
    prompt = PromptTemplate(
        input_variables=["query"],
        template="Generate Python code for: {query}"
    )
    chain = prompt | llm
    return chain.invoke({"query": query}).content

# Define the CodeReviewerAgent
def code_reviewer(code: str) -> str:
    llm = ChatOpenAI(model="gpt-3.5-turbo")
    prompt = PromptTemplate(
        input_variables=["code"],
        template="Review the following Python code and provide feedback:\n\n{cod
    )
    chain = prompt | llm
    return chain.invoke({"code": code}).content

# Define the Multi-Agent Workflow using LangGraph
def create_multi_agent_workflow():
    # Create a graph
    workflow = Graph()

    # Add nodes for each agent
    workflow.add_node("generate_code", code_generator)
    workflow.add_node("review_code", code_reviewer)

    # Define the edges (flow of the workflow)
    workflow.add_edge("generate_code", "review_code")

    # Set the entry point
    workflow.set_entry_point("generate_code")

    # Compile the workflow
    return workflow.compile()

# Run the Multi-Agent Workflow
def run_workflow(query: str):
    workflow = create_multi_agent_workflow()
    result = workflow.invoke({"query": query})
    return result

# Example Usage
if __name__ == "__main__":
    user_query = "a function to calculate the factorial of a number"
    print("Generating code and reviewing it...")
    output = run_workflow(user_query)
    print("Generated Code:\n", output["generate_code"])
    print("\nCode Review:\n", output["review_code"])
```

Structuring Multi-Agent Business Logic with Prompting Parameters

For Multi-Agent Systems to function effectively, business logic and input structures must be well-defined. This includes:

- **Prompting Parameters** — Inputs that guide AI Agent behavior, similar to API parameters.
- **Business Logic Encoding** — Defining rules that dictate how AI Agents interact with tools and memory.
- **Standardized Input Models** — Using structured frameworks such as Python's Pydantic to enforce input validation and ensure data integrity when passing information between agents.

The Evolution of RAG: Agentic RAG

While traditional RAG focuses on retrieving and augmenting information, **Agentic RAG** takes this a step further by integrating the retrieval process with AI Agent capabilities. In this advanced pattern, the system doesn't just retrieve information — it actively decides how to use it.

For example, an Agentic RAG system might:

- Dynamically select which knowledge sources to query based on the task.
- Refine its retrieval strategy in real-time using feedback from the LLM or user.

- Combine retrieved information with other tools (e.g., APIs or databases) to generate more comprehensive responses.

This approach is particularly useful in complex scenarios where static retrieval isn't enough. For instance, in legal research, an Agentic RAG system could retrieve case law, analyze precedents, and even suggest arguments based on the retrieved data.

Agentic RAG represents the next evolution of retrieval-augmented systems, further blurring the lines between retrieval and action.

Limitations of AI Agents

While AI Agents excel at decision-making, planning, and executing tasks, they are not without their challenges. Here are some of the key limitations:

- **Development Complexity**: Building AI Agents requires significant expertise in AI, machine learning, and software engineering. Designing agents that can plan, reason, and interact with multiple tools is inherently complex.
- **Tool Reliability and Integration Challenges**: AI Agents rely heavily on external tools (e.g., APIs, databases) to perform tasks. If these tools are unavailable, slow, or unreliable, the agent's performance can suffer. Connecting AI Agents to diverse tools and systems can be technically challenging, especially when dealing with proprietary or poorly documented APIs.
- **Context Window Constraints**: Like RAG, AI Agents are limited by the LLM's context window size. This restricts the amount of historical data or conversation context the agent can retain.

- **LLM Hallucinations:** AI Agents powered by LLMs can sometimes generate incorrect or nonsensical responses, especially when dealing with ambiguous or incomplete information.
- **Surface-Level Reasoning:** While AI Agents can simulate reasoning and planning, they lack true understanding or common sense. Their decisions are based on patterns in data rather than deep comprehension.
- **Brittleness**: AI Agents may struggle with tasks that require nuanced understanding or adaptability outside their training data.
- **Workarounds**: Developing standardized interfaces and protocols for tool integration can reduce complexity and improve reliability. With advances in LLMs with larger context windows (e.g., 100K+ tokens) can mitigate memory limitations. In the meantime, incorporating human oversight can help catch errors and ensure ethical decision-making. Furthermore, developing methods to make AI Agents' decision-making processes more transparent and interpretable.

So, What's the Bottom Line?

Knowing the difference between RAG and AI Agents is like choosing the right tool for the job.

- **RAG** gives LLMs a boost with extra knowledge, like adding research notes to a paper.
- **AI Agents** are more like project managers, making decisions and taking action.

Currently, ReAct (Reasoning and Acting) is a leading concept for AI Agents, enabling them to think through complex tasks and act strategically. But the future looks even more exciting. We're moving toward Multi-Agent Systems, where teams of AI Agents collaborate to tackle large-scale projects — think of it like a superhero squad!

Frameworks like LangGraph, CrewAI, AWS Bedrock, and Llama Index are already paving the way for this shift, and new, emerging frameworks are poised to push the boundaries even further. By structuring agent inputs with prompting parameters and validated schemas, we can create more reliable, scalable AI-driven applications.

Now, I'm curious — are any of you diving into Multi-Agent Systems? How do you think this will change the AI landscape? Share your thoughts or projects in the comments below — I'd love to hear what you're working on!

Here are some misconceptions about RAG and AI Agents:

Is RAG itself an agent?

- No. RAG is a retrieval-based augmentation technique, while AI Agents perform structured execution beyond retrieval. As a matter of fact, RAG can be used as one of AI-Agent's tools (more commonly used tools are APIs, but RAG is currently part of that tool set, a.k.a **Agentic RAG**

How do AI Agents decide which tools to use?

- Agents use predefined logic or reinforcement learning (e.g., ReAct framework, ToolFormer) to determine tool usage dynamically.

Why does LLM-generated output sometimes differ from retrieved documents?

- LLMs reframe, summarize, or hallucinate based on prompt design. Proper **retrieval validation** is critical.

Is history-aware retrieval the same as an AI Agent?

- No. While history-aware retrievers improve contextual grounding, they lack autonomous decision-making abilities.

Why do we need memory in AI Agents?

- To handle multi-turn queries effectively. Without memory, LLMs lose the conversation context.

How can we validate RAG-generated responses?

- To make sure RAG-generated responses are accurate and relevant, we can use metrics like retrieval precision, embedding similarity, and human checks. These help confirm that the information retrieved is on point and the final answer makes sense in context.

Is zero-shot reasoning sufficient for AI Agents?

- Not really. While zero-shot reasoning lets AI Agents handle tasks without prior examples, it often falls short in complex situations. For tasks that need multi-step planning or deeper context, AI Agents usually rely on memory or external reasoning tools to work effectively.