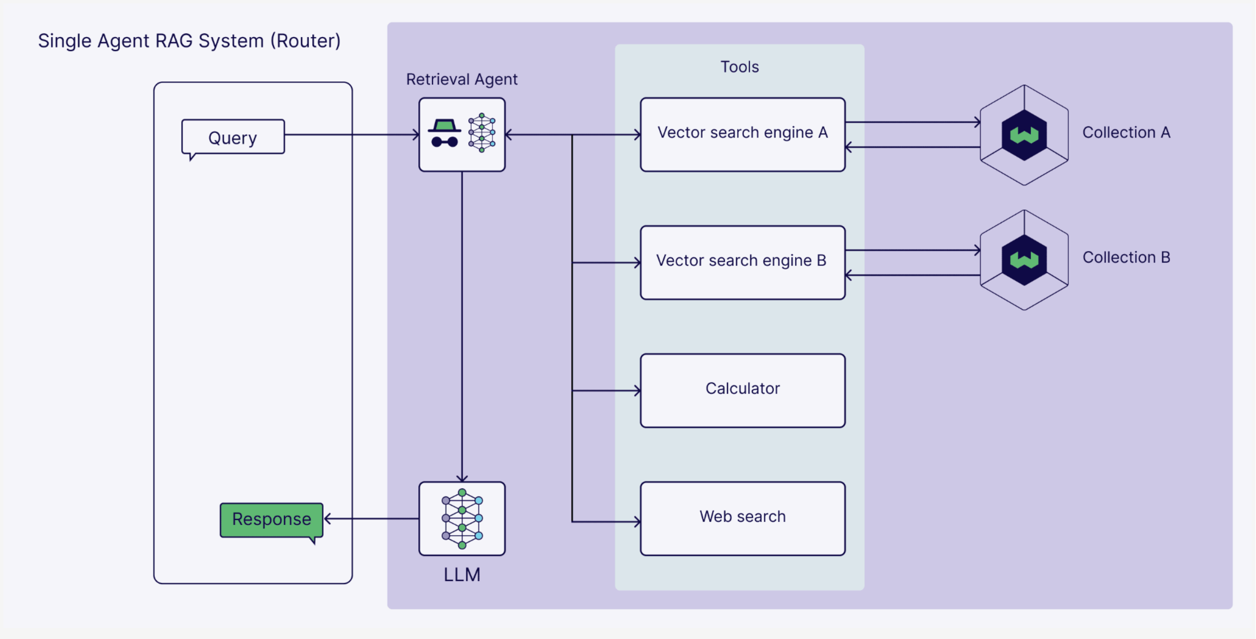
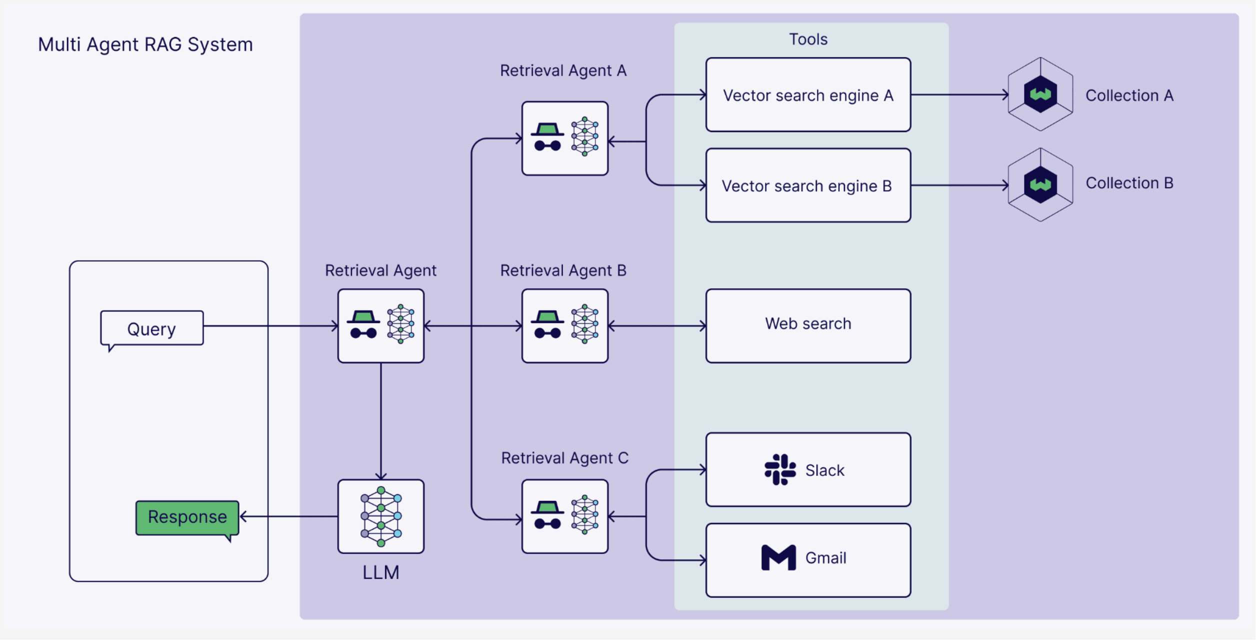
Agentic RAG

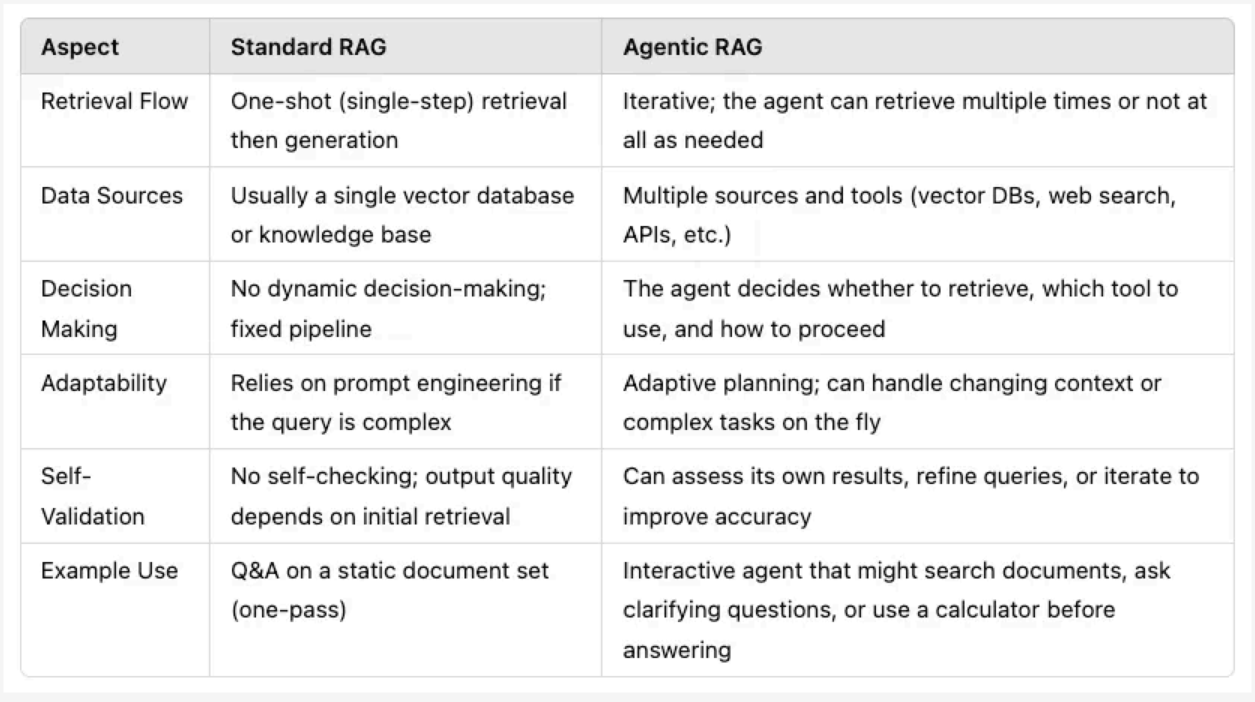
LLM with function calling vs AI Agents





Types of Agentic RAG Based on Function

1. **Routing Agent**
2. **One-Shot Query Planning Agen**
3. **Tool Use Agent**
4. **ReAct Agent (Reason + Act)**
5. **Dynamic Planning & Execution Agent**



Aspect	Standard RAG	Agentic RAG
Retrieval Flow	One-shot (single-step) retrieval then generation	Iterative; the agent can retrieve multiple times or not at all as needed
Data Sources	Usually a single vector database or knowledge base	Multiple sources and tools (vector DBs, web search, APIs, etc.)
Decision Making	No dynamic decision-making; fixed pipeline	The agent decides whether to retrieve, which tool to use, and how to proceed
Adaptability	Relies on prompt engineering if the query is complex	Adaptive planning; can handle changing context or complex tasks on the fly
Self-Validation	No self-checking; output quality depends on initial retrieval	Can assess its own results, refine queries, or iterate to improve accuracy
Example Use	Q&A on a static document set (one-pass)	Interactive agent that might search documents, ask clarifying questions, or use a calculator before answering

Reference:

1. https://ai.plainenglish.io/agentic-rag-how-autonomous-ai-agents-are-transforming-industry-d3e2723f51e8
2.

Agentic Retrieval-Augmented Generation (Agentic RAG): A Technical Deep Dive

Abstract:
Agentic Retrieval-Augmented Generation (Agentic RAG) is an emerging paradigm that integrates autonomous AI agents with retrieval-augmented generation pipelines. By combining the knowledge grounding of RAG with the decision-making and planning abilities of AI agents, Agentic RAG systems can dynamically retrieve information, use tools, and iteratively refine their outputs to tackle complex tasks. This whitepaper provides a comprehensive technical overview of Agentic RAG, targeted at engineering managers and software architects. We delve into its core architecture (planner, retriever, reasoning agent, tooling, memory), compare it to traditional RAG in a feature-by-feature matrix, explore use cases across industries, discuss current limitations (such as tool errors and latency), review leading tools and frameworks (LangChain, CrewAI, AutoGen, Semantic Kernel, DSPy), examine real-world case studies, and outline future research directions. The goal is to equip technical leaders with a deep understanding of Agentic RAG's capabilities, trade-offs, and implementation landscape, enabling informed decisions about adopting this technology in enterprise applications.

1. Introduction to Agentic RAG

Retrieval-Augmented Generation (RAG) is an AI framework that connects a generative model (like an LLM) with an external knowledge source to improve the factual accuracy of outputsibm.com. In a standard RAG pipeline, a user query is first used to retrieve relevant documents from a knowledge base (often via vector similarity search), and those documents are then provided as context to the LLM, which generates a responseibm.com. This approach grounds the model's answers in up-to-date or domain-specific data, reducing hallucinations and obviating the need for extensive fine-tuningibm.comanalyticsvidhya.com. Traditional RAG, however, is a *reactive*, single-pass process: it retrieves once and generates an answer, without the ability to plan multiple steps or use additional tools beyond the static knowledge basemedium.comglean.com. This limitation means that standard RAG can struggle with ambiguous or complex queries that require multi-step reasoning, query reformulation, or accessing diverse data sourcesweaviate.iomedium.com.

Agentic RAG represents the next evolution of this paradigm, introducing an *agentic layer* that endows the system with autonomy, planning, and tool useweaviate.ioibm.com. In an Agentic RAG system, an AI agent (or multiple agents) orchestrate the retrieval and generation process: deciding *what* to search, *when* to search, and *which tools or data sources* to consult at each stepglean.com. These agents are typically powered by LLMs themselves, augmented with a "reasoning policy" (often implemented via prompt engineering techniques like ReAct or function calling) that lets them perform complex, non-linear workflowsmedium.commedium.com. In effect, Agentic RAG turns a static QA system into an *autonomous researcher* that can iteratively ask follow-up questions, validate information, call external APIs, and only halt when a satisfactory answer is foundmedium.commedium.com.

Why now? The rise of Agentic RAG has been driven by the convergence of two trends: the success of RAG in grounding LLMs, and the emergence of AI agents capable of sequential decision-making. 2024 saw an explosion of interest in autonomous AI agents that can use tools and collaborate (e.g. ChatGPT's function-calling, AutoGPT, multi-agent frameworks), prompting researchers and practitioners to merge these capabilities with RAGanalyticsvidhya.comanalyticsvidhya.com. The result is a more adaptive form of AI system. Agentic RAG allows LLMs to conduct information retrieval from multiple sources and handle more complex workflows than traditional RAGibm.com. For example, instead of just looking up one document and answering, an Agentic RAG chatbot could decide to perform a web search, read the results, invoke a calculator for a computation, and cross-verify an answer – all autonomously – before responding to the userweaviate.ioglean.com. This ability to **"reason, plan, and act"** makes agentic systems especially powerful when dealing with ambiguous queries or tasks that require combining knowledge from various domainsibm.comglean.com.

In summary, Agentic RAG augments the classical retrieve-and-generate loop with an intelligent agent layer. This agent layer confers several key benefits: **adaptability** to the query's needs (through step-by-step planning and conditional logic), **flexibility** in accessing heterogeneous data/tools (beyond a single vector database) and **iterative refinement** of results (self-evaluating and re-searching as needed)glean.comibm.com. The following sections provide a deep dive into how Agentic RAG works, its architecture and components, and how it compares to vanilla RAG systems.

2. Core Architecture and Components

Agentic RAG architectures build upon the traditional RAG pipeline by embedding an agent (or agents) that control the flow of information between the user, the retrieval system, and the generative modelweaviate.ioweaviate.io. While implementations can vary, most Agentic RAG systems share a common set of **core components**:

- **Planner (Agent Controller)** – the decision-making module that plans the sequence of actions (e.g. whether to retrieve, which source to query, whether to use a tool, etc.). In many designs, this is an LLM itself acting as a *reasoning agent* with a special prompt.

- **Retriever (Knowledge Source)** – the mechanism for fetching external information (documents or data) relevant to the query (e.g. vector database search, web search API, SQL query).

- **Reasoning Agent (LLM)** – the large language model that carries out the reasoning steps, guided by the planner. Often the planner and reasoning LLM are the same underlying model or tightly integrated.

- **Tooling Layer (Action Interfaces)** – a set of tools or APIs the agent can invoke to perform tasks beyond text generation, such as running code, doing math, querying databases, sending web requests, etc.

- **Memory (State Management)** – stores context from previous interactions or intermediate steps, enabling the agent to maintain state across multi-step reasoning.

These components work together in a loop, orchestrated by the agent's logic. Below, we examine each component in detail and how they interconnect in an Agentic RAG system.

Planner (Agent Controller)

The **planner** is the "brain" of the agentic system – it is responsible for interpreting the user's query, devising a strategy to answer it, and deciding on actions to take. In practical terms, the planner is often an LLM prompt that encourages the model to think step-by-step (using a method like ReAct, CoT, or a domain-specific policy)weaviate.iomedium.com. For example, using the ReAct frameworkweaviate.io, the agent prompt template might instruct: *"Think about what to do next, then either say an action or give the final answer."* This turns the LLM into a **Reason+Act** agent that can alternate between reasoning ("Thought…") and acting ("Action: search tool with query X")weaviate.ioweaviate.io. The planner monitors the agent's observations from each action and decides the next step until the task is completeweaviate.ioweaviate.io.

Key capabilities of the planner include **query routing, step-by-step reasoning, and decision-making**ibm.com. On receiving a complex query, the planner might break it into sub-problems or choose which knowledge source is most appropriateweaviate.ioglean.com. For instance, consider a question: *"Summarize the impact of new EU banking regulations on US fintech startups."* A static RAG would do a single vector search on a fixed corpus. An Agentic RAG's planner, however, might decide: (1) search an internal compliance database for "EU banking regulations summary", (2) search news APIs for "US fintech response to EU regs", and (3) call a finance analysis tool to interpret the findings. The planner's logic enables this multi-step approach by dynamically adjusting the plan based on intermediate results. If one source yields insufficient info, the planner can branch out – **e.g. try a web search if the internal DB had gaps**weaviate.ioglean.com. This adaptability is a hallmark of Agentic RAG.

In more advanced setups, the planner might itself be composed of multiple sub-agents or a hierarchy (a "manager" agent delegating to specialist agents – see multi-agent systems in Section 7). But in a single-agent architecture, it's essentially the loop that drives the agent's **Thought → Action → Observation** cycleweaviate.iomedium.com. The outcome of the planning component is a series of actions: for example, *Action 1: use Retriever with query Q1; Action 2: call Calculator on retrieved data; Action 3: when satisfied, output answer*. These actions are executed in the tooling layer, and the resulting observations (retrieved text, calculation result, etc.) are fed back into the planner's context (often appended to the LLM's dialogue or scratchpad)weaviate.iomedium.com. The loop continues until the planner signals that an answer can be formulated (or a maximum number of iterations is reached for safety).

In summary, the planner imbues the system with **autonomy and intelligence in control flow**. It transforms the retrieval process from a one-shot lookup into an *iterative search-and-think* mission. As noted by IBM, this is a shift “from static rule-based querying to adaptive, intelligent problem-solving”ibm.com. The planner’s reasoning ensures that each query can trigger a customized workflow tailored to the problem’s complexity.

Retriever (Knowledge Source)

The **retriever** in Agentic RAG serves the same fundamental purpose as in traditional RAG: fetching relevant information to ground the model’s responses. This component typically comprises an embedding model and a vector store (for similarity search), or other search indices and APIsweaviate.io. What changes in the agentic paradigm is *how* the retriever is used and the diversity of retrieval options under the agent’s control.

In a vanilla RAG, there is usually a single retrieval step from a single corpus (e.g. a vector database of company documents). Agentic RAG generalizes this – an agent may have access to **multiple retrievers** and decide which to use (or use them sequentially) based on the queryweaviate.ioweaviate.io. For example, an agent might choose between: a corporate wiki vector search, a web search tool, and a relational database query. Each of these is a “retriever” in a broad sense. The agent can even invoke them one after another if needed, effectively performing *multi-hop retrieval* across systemsglean.commedium.com. This addresses a key limitation of basic RAG, which only considers one source and does no validation of retrieved infoweaviate.ioweaviate.io. By having an agent orchestrate the retrievers, Agentic RAG can retrieve iteratively: first gather some context, then decide if additional or alternative retrieval is needed, possibly reformulating the query and searching againglean.comweaviate.io.

A simple Agentic RAG might incorporate at least two knowledge sources to demonstrate value – for instance, a **Single-Agent RAG Router** that chooses between a private knowledge base and a web search depending on user query typeweaviate.io. Weaviate describes this router scenario: *"you have at least two external knowledge sources, and the agent decides which one to retrieve from"*weaviate.io. If a question is about internal policy, hit the vector DB; if it's about current events, do a web search. This conditional routing is a form of agentic decision-making at the retrieval layer, improving accuracy by using the right tool for the jobweaviate.ioweaviate.io.

Beyond choosing sources, the agent can also **formulate the queries** that go into the retrievers. Instead of using the user's query verbatim for vector search, it might rephrase or generate targeted keywords (similar to the ReAct style where the agent decides the search query). This is important for complex questions where the initial query might be too broad or vague. The agent's ability to analyze the user prompt and generate a good search query is effectively query expansion or reformulation – an intelligent retriever usage that boosts RAG performanceglean.comarxiv.org. For example, user asks: "What are the health implications of microplastics, and have there been any policy responses in 2025?" The agent could break this: query 1 to a scientific papers index: "microplastics health effects 2023 study", query 2 to news search: "2025 policy microplastics ban". Traditional RAG wouldn't do this split; an agentic retriever does, yielding far more relevant context for each sub-part.

Finally, after retrieval, the agent doesn't blindly accept the results. Part of the Agentic RAG loop is **verification and potential re-retrieval**. If the retrieved documents seem irrelevant or insufficient, the agent can discard them and try a different approachweaviate.io. Some advanced implementations include a *Relevance Evaluation Agent* that checks retrieved docs and triggers a correction cycle if neededarxiv.orgarxiv.org. We will discuss such "corrective RAG" patterns later; the key point is that retrieval in Agentic RAG is not a one-and-done step, but a *managed, dynamic process*. The agent's reasoning layer tightly integrates with the retrieval module to ensure high-quality information is gathered. In essence, **the retriever becomes agentic** – it's used in a flexible manner under an agent's directionweaviate.ioweaviate.io.

Reasoning Agent (LLM)

At the heart of the system lies the **reasoning agent**, usually implemented by a Large Language Model (LLM) like GPT-4, PaLM, or an open-source model, which interprets the context and produces conclusions or answers. In many designs the reasoning agent and planner are one and the same (the LLM both plans and generates text), but conceptually we can distinguish the *planning function* (decision-making) from the *generation function* (synthesizing an answer or intermediate result). The reasoning agent is what ultimately generates the content – whether it's an answer to the user, a reformulated query, or a summary of retrieved info.

In the **ReAct** paradigm of agents, the LLM interweaves reasoning thoughts and actionsweaviate.io. Those “thoughts” are the reasoning agent at work: the LLM analyzing the question, the retrieved documents, and deciding next steps. For example, it might produce a thought: *“The query asks for 2024 Euro winner details, I should use web search.”* (as per the workflow example in Analytics Vidhyaanalyticsvidhya.com). The LLM then outputs an action (search), gets results, and then the reasoning agent part kicks in again to read those results and integrate them into a final answer. When sufficient information is gathered, the agent transitions from action mode to answer mode, and the LLM generates the **final response** for the useranalyticsvidhya.com.

Crucially, the reasoning agent brings in the LLM’s powerful capabilities for **natural language understanding and generation**. It can synthesize information from multiple documents into a coherent explanation, perform on-the-fly translations or calculations if needed, and produce well-structured outputs (e.g. an email draft, a step-by-step plan, or a plain answer). This component is what ensures the output is not just a collection of snippets but a useful **synthesis of knowledge plus reasoning**arxiv.org. In our earlier example of summarizing microplastic health impacts and policies, after using the retriever to get both scientific studies and news articles, the reasoning LLM would weave together a summary: *“Research shows microplastics are linked to hormonal and digestive issues in marine life and potentially humansarxiv.org. In 2025, the EU implemented a policy banning certain plastics in response to these findings, and the US FDA is evaluating similar measuresglean.com…”*. The agent’s answer isn’t copy-pasted – it’s generated with understanding.

Memory (discussed below) plays a role in the reasoning process by providing the agent with context from previous steps or interactions. The LLM can refer to what it has done so far. For instance, after step 1 retrieval, the agent's memory holds those results; in step 2, the LLM can say "Based on the earlier info, it looks like X, now I will do Y". This chain-of-thought is preserved, giving consistency to the reasoning.

One challenge the reasoning agent addresses is making sure the answer actually uses the retrieved evidence correctly, rather than hallucinating. By design, RAG aims to reduce hallucinations, but if the retrieved info is tangential or insufficient, even a smart agent can generate a plausible-sounding yet incorrect answer. Agentic RAG mitigates this via iterative checking: the reasoning agent can *self-critique* its draft answer and decide to retrieve more if uncertainglean.commedium.com. This self-reflection is often prompted in the LLM (like "Check if all aspects of the query are answered. If not, search again."). According to Medium's analysis, an agentic RAG LLM can include an **auto-validation loop**, where it verifies if the question was fully answered or if gaps remain, and then act accordinglymedium.com. This leads to higher accuracy over multiple iterations.

In multi-agent configurations, there may be several reasoning agents specialized for different tasks (e.g. one agent LLM handles database queries, another handles text summarization). They communicate either through a shared memory or by calling each other as toolsmedium.com. But even then, each agent is fundamentally an LLM reasoning about its sub-problem. For most single-agent systems, we simply have one LLM that does it all: it plans, it reads retrieved data, and it generates the answer. The *prompt design* for this LLM is critical – it must be provided with instructions on how to act (planner role) and how to format reasoning vs answers (often using a few-shot example of the Thought/Action/Observation loop)weaviate.io. When implemented well, the reasoning agent exhibits an almost human-like approach: interpret question, gather info, analyze, verify, conclude.

To summarize, the reasoning agent is the **analyst** and **report writer** of the Agentic RAG pipeline. It uses the data pulled by the retriever, follows the plan or dynamically updates it, and ultimately delivers the solution. It is central to the agent's ability to solve complex queries that require actual understanding and not just lookup. In the words of one LinkedIn article, this is where AI goes "from a passive assistant into an active collaborator that actively decides what to do next"linkedin.comlinkedin.com – the LLM isn't just answering, it's collaborating with the tools and data to produce the answer.

Tooling Layer (External Tools and Actions)

A defining feature of Agentic RAG (as opposed to vanilla RAG) is the integration of a **tooling layer** – a set of external tools or APIs the agent can invoke to perform actions. Traditional RAG is limited to retrieving textual context; Agentic RAG generalizes this by allowing arbitrary actions in service of answering the queryweaviate.ioglean.com. Tools can include: search engines, databases, calculators, code interpreters, email or messaging APIs, custom business applications, or even other AI models. This effectively extends the agent's capabilities beyond what the base LLM can do by itself.

For example, if a question requires a mathematical calculation or executing some code logic, an agent can call a Python interpreter tool, run the code, and use the resultarxiv.org. If the query is about current weather, an agent could call a Weather API rather than relying on potentially outdated training data. OpenAI's introduction of function calling in mid-2023 made tool integration with LLMs much more straightforward – the model can output a JSON object specifying a function name and arguments, which the application executes and returns the result back to the modelweaviate.ioweaviate.io. This mechanism has been rapidly adopted to plug LLMs into calculators, web browsers, and more. As Weaviate notes, by using function calling or agent frameworks, "developers quickly started building applications that plugged GPT-4 into code executors, databases, calculators, and more"weaviate.ioweaviate.io.

In Agentic RAG specifically, common tools include: **Vector search query engines** (to perform the RAG part), **web search** (to fetch live information)weaviate.io, **internal APIs** (for e.g. querying a CRM or ticketing system), **SQL database connectors**, **calculators or units converters**, **document parsers**, and so onweaviate.io. Essentially, the agent's action space is not restricted to "search knowledge base" – it can be any well-defined operation that might help answer the question. The architecture diagram in Medium's article illustrates an agent with multiple tools at its disposal in a loop: a vector DB lookup, a web search, and "other APIs," all of which the agent can choose from as neededmedium.commedium.com.

The agent's prompt or policy typically includes descriptions of available tools and how to invoke them. For example, a tool might be described as `search_tool(query): searches the web for the query` or `sql_tool(query): executes an SQL query`. The LLM then can output something like: `Action: search_tool("latest EU banking regulations 2024")`. The execution framework catches this, performs the action, and feeds the result (maybe the top web snippet or database rows) back to the LLM as an observationmedium.commedium.com.

One crucial tool for RAG is still the vector database itself – in Agentic RAG, the vector DB lookup becomes just one of many possible actions. The agent might first do a web search, then vector search on internal docs, then combine the results. In that sense, the boundary between "retriever" and "tool" is blurry; a retriever is essentially a specialized tool. We treat the *tooling layer* as encompassing all these actions. Weaviate specifically enumerates that a retrieval agent could have tools such as: "*Vector search engine, Web search, Calculator, Any API (email, chat programs, etc.)*"weaviate.io. This list shows that RAG can expand to full-fledged API orchestration. For instance, an enterprise Agentic RAG might even integrate with an internal Slack API to fetch recent discussions if relevant to a queryweaviate.ioweaviate.io.

The addition of tools greatly enhances what tasks the agent can automate. Use cases like data analysis pipelines (where the agent might query a database and then run computations) or multi-modal tasks (where an agent might call an image analysis API or text-to-speech) become feasible. One concrete example given by Glean: an agent answering "Summarize this contract and flag any policy violations" would use multiple tools – one to extract clauses from the contract PDF, another to retrieve the relevant company policies, and perhaps a comparison scriptglean.com. The agent then combines those outputs to produce a final summary highlighting the violations. Without tool use, an LLM wouldn't be able to parse a PDF or know company policies not in its training data. Agentic RAG fills that gap with its tooling layer.

From an engineering perspective, integrating a tooling layer requires careful definition of interfaces and robust error handling. Each tool should be deterministic and give some output the LLM can understand (or an error if it fails). A known challenge is **tool execution errors** – if a tool fails (e.g., API timeout, or the LLM gave it malformed input), the agent needs to handle that gracefully. Many frameworks (like LangChain Agents or Microsoft's Semantic Kernel) provide abstractions to define tools and automatically parse tool invocation outputs. For example, the Weaviate blog demonstrates defining a `get_search_results(query: str) -> str` function that queries Weaviate's database and returns results as a formatted stringweaviate.ioweaviate.io. This function is then added to the model's allowed tools via a schema, enabling the model to call `get_search_results` when neededweaviate.io. By designing tools with clear inputs/outputs and perhaps constraints (like maximum returned text length), the agent's interactions with them can be made reliable.

In sum, the tooling layer transforms an Agentic RAG agent from just a *reader* into an *actor*. It can take steps in the real or digital world to gather information and solve sub-tasks. This significantly **extends the scope** of problems that can be tackled. Rather than being limited to the contents of a single knowledge base, the agent becomes a general problem-solver – it can retrieve from various sources, calculate, transform data, and even trigger actions (like sending an email response, if that were a tool) as part of fulfilling a user's requestweaviate.ioglean.com. Section 4 (Use Cases) will highlight how such tool use enables applications like autonomous research or data pipeline orchestration that would be impossible with a static RAG approach.

Memory (Short-term and Long-term)

Memory is a critical component that gives Agentic RAG its context-awareness and continuity. We can think of memory in two forms: **short-term memory**, which holds the state of the current agent dialog or reasoning chain, and **long-term memory**, which stores knowledge or experiences over multiple sessions. Both types allow the agent to "remember" information so it can use it in reasoning or future decisions.

In a single query session, the agent's short-term memory might include: the original question, the chain-of-thought so far (all thoughts, actions, observations in the loop), and any intermediate results or conclusions drawn. This is often stored as part of the LLM's context (e.g., appended to the prompt token sequence) or in a structured state that the agent can query. By maintaining this state, the agent ensures consistency and can refer back to earlier steps ("As found above, X is true, so now we check Y"). It's essentially the **"scratchpad"** where the agent accumulates partial answers and evidence. The ReAct framework explicitly relies on this: the transcript of Thoughts/Actions/Observations is the LLM's memory of what it has doneweaviate.io. This prevents the agent from repeating the same search twice or forgetting why it took a step. It also allows multi-step reasoning that feels coherent and integrated.

Long-term memory refers to storing information across interactions or for use in future queries. For example, an agent might cache the results of expensive operations or remember a user’s preferences. In RAG contexts, long-term memory could be implemented as a **knowledge base of previous Q&A pairs or summary of past discussions** (sometimes called semantic caching). IBM notes that agentic RAG systems often use semantic caching to store previous queries, contexts, and resultsibm.com. This means if a similar question arises later, the agent can recall the answer without recomputing, or use past findings to inform current reasoning. Memory can also be domain-specific: for instance, an agent might build up a fact base of key facts it has verified, to avoid re-verifying them every time (this borders on continual learning).

Memory is closely tied to the **“statefulness”** of agentic systems. Unlike stateless RAG (each query independent), an agent can carry state. One example is in a multi-turn conversation: an Agentic RAG chatbot assisting with research might keep track of the conversation so far and the documents already retrieved, to avoid redundancy and drive the conversation forward. Short-term memory here might be the conversation history (like ChatGPT does) plus the chain-of-thought not shown to the user. Long-term might be a persistent profile of the user’s knowledge if the system is used repeatedly.

The architecture must manage memory carefully to avoid overwhelming the LLM context window. Some implementations summarize or compress old steps. Others may store detailed state in a vector DB and let the agent retrieve from its own memory when needed (a form of *introspective retrieval*). For instance, if an agent solved a similar task yesterday, it might vector-search its memory of that session's notes to get a head start on today's problem.

Memory also contributes to the agent's ability to do **reflection and self-correction**. After completing a task, the agent might store the solution and a self-evaluation of it. Later, if it encounters a related problem, it can reflect on how it solved it before. This is part of proposed agentic patterns like "self-reflection" agentsanalyticsvidhya.com. One can consider memory as the implementation of the agent's "experience."

In practice, frameworks like LangChain and Semantic Kernel provide constructs for both short-term (context windows, transient in-memory variables) and long-term memory (vector stores for past interactions). For example, Semantic Kernel's planner can use the *Memory* API to recall previous facts or conversations when planning new stepsibm.com. Another concrete approach is **semantic caching** mentioned by IBM: store recent Q&As so that if the same query comes, the system can answer immediatelyibm.comibm.com. This not only speeds things up but also ensures consistency in answers.

From the architecture viewpoint, memory is often depicted as a store the agent can read from and write to. In the Weaviate blog, memory is listed as one of the core components of an AI agent, with both short-term (recent interactions) and long-term (knowledge accumulated)weaviate.ioweaviate.io. Analytics Vidhya similarly highlights memory as *"the key to contextual intelligence,"* breaking it into short-term, long-term, and semantic memory for general knowledgeanalyticsvidhya.comanalyticsvidhya.com. By maintaining short-term context, the agent gives coherent answers that account for the conversation or multi-step context. By leveraging long-term memory, it can improve over time and handle **context carry-over** across sessions.

To illustrate, imagine an **autonomous research agent** that is doing an extended literature review over days. It might have a long-term memory of which papers it has analyzed and what their key points were. When asked a new question on day 2, it can recall that memory instead of re-reading all papers, focusing only on new information. This behavior mimics a human researcher who remembers yesterday's findings. Agentic RAG aims for such continuity, which is crucial in large projects or dialogues.

In summary, memory in Agentic RAG provides **contextual continuity** and **learning capability**. It ensures the agent's behavior is not myopic to just the immediate query. By storing and retrieving relevant context, the agent delivers more context-aware responses and can optimize its workflow (no need to redo what it already knows). Memory modules – whether as part of the LLM's context or external databases – are thus foundational to scaling agentic systems to complex, long-running tasks and dialoguesibm.comanalyticsvidhya.com.

Having examined the core components (Planner, Retriever, LLM Reasoning, Tools, Memory), we can see how they interlock to form an Agentic RAG system. The typical control flow is: **User Query → Planner decides action → Tool executes (retriever is a common tool) → Observation returned → (Planner/LLM) updates memory and decides next action → ... → Finally, LLM produces answer**. This loop may repeat many times with different tools. The inclusion of planning, tool use, and memory is what distinguishes this architecture from a basic RAG pipeline. In the next section, we will explicitly compare Agentic RAG to traditional RAG to highlight these differences.

3. Comparison with Traditional RAG

Agentic RAG extends traditional RAG with new capabilities. It is useful to compare them feature by feature to understand the benefits and trade-offs. **Traditional RAG** (sometimes called "vanilla RAG") refers to the standard retrieve-and-generate approach without an agent orchestrating multiple stepsmedium.com. **Agentic RAG** adds the agent layer as described. The table below summarizes key differences:

Feature	Traditional RAG	Agentic RAG
External Knowledge Sources	Single source (one vector index or corpus)ibm.com.	Multiple sources (can query various DBs, web, APIs)ibm.comibm.com.
Tool Use Beyond Retrieval	No tool use – only retrieves text for contextweaviate.io.	Yes, can invoke external tools/APIs (web search, calculators, custom functions)weaviate.ioglean.com.

Workflow	One-shot, fixed sequence: retrieve → generatemedium.com.	Iterative, flexible sequence with conditional loopsmedium.comglean.com.
Decision-Making	Reactive (no decision, always does the same steps)medium.com.	Autonomous planning: agent decides whether/what to retrieve, when to stop, etc.medium.comglean.com.
Query Refinement	No query reformulation (uses user query directly)medium.com.	Agent can reformulate queries or do multi-hop queries based on intermediate resultsglean.comarxiv.org.
Context Validation	No self-check – uses whatever was retrieved without verificationibm.com.	Agent can validate retrieved info, discard irrelevant data, or re-search if neededglean.comarxiv.org.
Adaptability	Static – cannot change strategy if answer is incompleteibm.com.	Adaptive – agent monitors progress and can change strategy or tools on the flymedium.comibm.com.

Collaboration	Single-step, no notion of collaboration.	Supports multi-agent collaboration (agents can delegate tasks or check each other)ibm.commedium.com.
Memory Usage	Limited to prompt context (usually just retrieved docs)weaviate.io.	Maintains state across steps; can incorporate short-term and long-term memory for contextweaviate.ioanalyticsvidhya.com.
Handling Complex Queries	Often struggles if query requires multi-step reasoning or combining infoarxiv.orgarxiv.org.	Designed for complex, multi-part queries; excels at multi-hop reasoning tasksarxiv.orgmedium.com.
Accuracy & Completeness	May give incomplete or partly incorrect answers if single retrieval misses something (no self-correction	Tends to higher accuracy by iterative improvement; agent can fill gaps by additional retrieval or verificationibm.commedium.com.

	loop)arxiv.orgarxiv.org.	
Latency	Lower latency (one LLM call after retrieval) – fast responses.	Higher latency due to multiple steps and tool calls; agent loops incur overheadarxiv.orgarxiv.org.
Scalability	Scales with database size (may slow if corpus large, but only one pass)arxiv.org.	Can handle multi-domain queries via specialized agents, but more computationally heavy for high volume queriesarxiv.orgarxiv.org.
Factuality	Grounds answers in retrieved text, reducing hallucinations (but if retrieval fails, it can hallucinate)ibm.comarxiv.org.	Further reduces hallucinations via tool use and validation; however, agent could propagate errors if a tool returns false info (hallucination can still occur across steps).
Development Complexity	Simpler to implement (essentially two components:	More complex (requires implementing agent logic, tool interfaces, error handling for

	retriever + LLM)weaviate.io.	multi-step)arxiv.orgarxiv.org.

Table: Traditional vs Agentic RAG. (Agentic RAG introduces autonomy, multi-step reasoning, and tool integration, at the cost of complexity and runtime overhead.)

As the table highlights, Agentic RAG **generalizes** RAG along multiple axes. A useful analogy from Weaviate is: *"Common (vanilla) RAG is like being at the library (before smartphones) to answer a specific question. Agentic RAG is like having a smartphone with a web browser, a calculator, your emails, etc."*weaviate.io. In other words, traditional RAG can look something up in one place, whereas agentic RAG can leverage an entire toolbox of resources in real-time.

From a capability standpoint, the biggest advantages of Agentic RAG are **flexibility, adaptability, accuracy, and scalability to more complex tasks**ibm.comibm.com:

- **Flexibility:** Agentic RAG can pull data from multiple external knowledge bases and use external tools, whereas standard RAG is usually tied to one datasetibm.com. It's not limited to text corpora – it can incorporate structured data, perform computations, or call any API needed. This broadens the scope of queries it can handle (for example, a single agent could answer "What's our current sales vs target? Plot a graph" by querying a sales database and invoking a plotting function – beyond a pure text answer).

- **Adaptability:** Traditional RAG is static; it doesn't change its behavior between queries except through prompt engineering. Optimal results might require the user or developer to manually tune prompts for each scenarioibm.com. Agentic RAG is more *intelligent* – it observes intermediate results and adapts. If initial results are irrelevant, it can try a different approach. It can handle changing requirements on the fly (e.g., if mid-way the user adds a clarification, the agent can incorporate it). In essence, it moves from "lookup" to **problem-solving**. Multi-agent systems even allow agents to verify each other's work, introducing a form of built-in quality control and adaptability to errorsibm.com.

- **Accuracy:** By iterating and self-correcting, Agentic RAG can achieve higher answer accuracy for difficult queries. Traditional RAG has no mechanism to verify it found the right info – it might return whatever was top-ranked, which could be wrong or insufficientibm.com. An agent can notice "I don't have a complete answer yet" and keep diggingmedium.com. It also can mitigate errors like hallucinations by grounding itself multiple times. That said, if not carefully designed, an agent might also propagate a hallucination (for instance, if the LLM incorrectly "thinks" some fact and doesn't double-check it). But generally, the opportunity for iterative refinement leads to better resultsibm.com. In benchmarks, agentic strategies have shown improved factuality and completeness, especially on multi-hop questionsarxiv.org.

- **Handling Complexity:** Standard RAG works well for straightforward Q&A where one pass sufficesmedium.com. It falls short on complex queries that require reasoning through multiple pieces of information. Agentic RAG shines in those scenarios – it was *created* to tackle themmedium.comarxiv.org. This includes tasks like planning (the agent can break a task into steps) and decision support (the agent can weigh information, check consistency, etc.). Essentially, Agentic RAG can solve problems that you would otherwise need a human analyst or multiple distinct queries to solve.

- **Multimodality:** Traditional RAG and LLMs historically dealt only with text. Agentic systems, especially with new multimodal LLMs (like GPT-4 Vision or others), can incorporate images, audio, or other data types via tools. IBM notes that agentic RAG benefits from advancements in multimodal LLMs to work with a greater range of data – e.g., analyzing images or audio transcripts in addition to textibm.com. If an LLM can interpret an image, the agent could use a camera or image tool as part of its pipeline. This is a relatively new development, but it further differentiates agentic systems from vanilla RAG.

On the other hand, **Agentic RAG introduces complexity and overhead**. Not every application needs an agent; for simple factoid questions, a plain RAG (or even just the LLM itself if knowledge cutoff isn't an issue) might be sufficient and more efficient. The agentic approach can be overkill if not needed. Key trade-offs/disadvantages include:

- **Higher Latency:** As noted, multiple steps mean more time. A single query might involve several LLM calls (thoughts and actions) plus external API calls. This could be seconds or tens of seconds, whereas a single RAG call might be a fraction of a second for retrieval plus a couple seconds for generation. If the task is time-sensitive and straightforward, the agent might be unnecessarily slow. Techniques like limiting max steps or using faster models for tool selection can mitigate this, but latency is inherently higher.

- **Increased Cost:** Each LLM invocation costs tokens (in API usage or compute). Agentic loops consume more tokens (chain-of-thought logs, multiple queries, etc.). Also, using external tools might have costs (e.g., API fees for web search or database queries). So answering one question with an agent might cost an order of magnitude more in API calls than a single RAG call. For enterprise deployments at scale, this is an important consideration.

- **System Complexity:** Building and maintaining an Agentic RAG system is more complex than a standard RAG pipeline. There are more moving parts: tool integrations, error handling (what if a tool fails or returns nothing?), prompt designs for the agent, and potential edge cases where the agent might loop indefinitely or take a suboptimal path. It requires more rigorous testing to ensure reliability. This complexity can also introduce new failure modes – for example, an agent might decide on a wrong sequence of actions that leads it astray, whereas a simple RAG either finds something or not but doesn't "get creative" in the wrong direction. Thus, developing agentic systems may need more expertise and robust frameworks.

- **Resource Utilization:** The agent's ability to scale to very high query volumes is somewhat limited by the heavier per-query computation. If you have millions of simple queries (like a search engine), a static retrieval pipeline is much cheaper and faster. Agentic RAG might be reserved for cases where queries are complex but fewer. However, one could design a hybrid: use a cheap classifier to detect if a query needs agentic treatment, otherwise answer with standard RAG, thereby optimizing resources.

Despite these trade-offs, the trend in industry is toward leveraging agentic capabilities where the problem demands it. As one blog put it, *"Agentic RAG marks a shift from merely searching for data to actively engaging with it in meaningful ways."*moveworks.com. It is not meant to replace simple RAG in all cases, but to **augment** what AI systems can do. Many enterprise scenarios that were once too complex for automation (involving multi-step research or workflow) are now within reach using Agentic RAG, albeit with careful engineering.

To concretely illustrate the difference: imagine asking a system *"Find any discrepancies between the data in Report A and Report B, and draft an email to the team explaining the findings."* A traditional RAG system cannot do this – it could maybe retrieve content of the reports, but it won't on its own perform a comparison or draft an email. An Agentic RAG system could: use a document reader tool to extract data from Report A and B, use the LLM to identify discrepancies, then use a template to draft an email summary, perhaps even sending it via an email API. The agent might break this down into multiple steps and ensure the final output is correct. This example showcases a qualitative leap in capability.

In conclusion, Agentic RAG maintains the original promise of RAG (grounding LLMs in external knowledge) while dramatically extending the complexity of tasks that can be automated. It **bridges the gap between question-answering and autonomous task completion**. The next sections will explore concrete use cases (Section 4) where this added capability unlocks new applications, as well as the limitations to be mindful of (Section 5).

4. Use Cases for Agentic RAG

Agentic RAG opens up a range of advanced applications across different domains. Here we discuss several use case categories where the technology proves especially valuable, often surpassing what either standalone LLMs or traditional RAG systems can achieve. These include: autonomous research agents, data pipeline orchestration, enterprise knowledge management, financial analysis, and legal document summarization. In each scenario, we'll describe how Agentic RAG is applied and why its agentic features are necessary.

Autonomous Research Agents

One exciting use case is an **autonomous research assistant** – an AI agent that can gather information on a given topic, synthesize findings, and even generate reports or recommendations without continuous human guidance. Researchers or analysts spend significant time formulating queries, reading numerous sources, and integrating the information. An Agentic RAG system can automate parts of this process by intelligently searching and reading on behalf of the user.

For example, consider a product manager who wants a competitor analysis: *"Research the top 5 competitors in our industry and summarize their recent strategies."* An autonomous research agent could: perform web searches for each competitor, retrieve news articles or financial reports, use the LLM to extract key strategy mentions, and compile a structured summary. This involves multiple steps (one per competitor, plus cross-comparison) that a single QA response cannot handle. Agentic RAG shines here by breaking the task down. The agent might spawn sub-queries like "Competitor X 2024 strategy announcement" and "Competitor X financial results Q3 2024" and loop through each competitor. Tools like web search APIs and perhaps a PDF reader (to parse financial reports) would be employed. The reasoning LLM would then integrate all this into a cohesive report.

Such an agent essentially functions as an **autonomous research analyst**. It can be interactive as well – the user could refine the request ("focus more on marketing strategies and less on product features") and the agent will adapt its search and analysis. Multi-agent setups can be used: for instance, one agent could focus on gathering facts while another agent focuses on writing the summary. They might exchange information, with the writer agent requesting more details if needed – mimicking how a junior researcher might gather data for a senior analyst to write up.

Academic research is another area. One could task an agent: *"Literature review on the effects of microplastics on human health."* The agent can query academic databases (using tools to interface with something like Semantic Scholar or arXiv APIs), retrieve relevant papers, perhaps use a summarization tool on each, store the key findings in memory, and then have the LLM produce a synthesized literature review. This is an extension of RAG (which would just answer a specific question from the literature) to a **proactive, goal-driven research process**. Notably, digital libraries are huge, so an agent might need to follow references or search iteratively (find a key paper, then search for papers citing that paper, etc.). Agentic RAG's iterative retrieval is well-suited for this kind of explorationarxiv.orgarxiv.org.

A concrete example from the literature: the *DigitalOcean article on Agentic RAG* presents how merging RAG with agent decision-making is crucial as tasks become more complex and require knowledge sharingdigitalocean.comdigitalocean.com. It notes that as tasks increase in complexity, relying on only RAG or only agents alone "may not be enough," hence the merge into Agentic RAGdigitalocean.com. Autonomous research is exactly this scenario – complex multi-step information gathering that benefits from an agent's reasoning.

Another manifestation of a research agent is in **scientific discovery or literature analysis**. The arXiv survey on Agentic RAG highlights applications in healthcare and educationarxiv.orgarxiv.org, where an agent might help researchers keep up with new publications or help students by autonomously finding and explaining learning resources. For instance, an educational agent might take a topic (e.g., "quantum computing basics"), search for explanatory materials, fetch relevant textbook sections or tutorial blogs, then generate an easy-to-understand summary or lesson plan. It can also answer follow-up questions by pulling more info as needed – effectively acting as a tutor that has the whole internet or library at its disposal.

In summary, autonomous research agents leverage Agentic RAG to conduct non-trivial information gathering and analysis tasks end-to-end. The **agentic features (planning, multi-hop search, memory)** are what enable a single AI agent to persist over a long research session and produce a meaningful outcome. This has applications in market research, competitive intelligence, academic literature reviews, patent research, and any scenario where the query is more like *"Investigate X and report back"* rather than a one-sentence question.

Data Pipeline Orchestration

Agentic RAG can also serve in **data pipeline orchestration and analysis** tasks. This refers to scenarios where an AI agent coordinates multiple steps in a data processing workflow – for example, retrieving data from a database, performing transformations or analysis, and generating reports or triggering actions based on the results. Traditional RAG is not sufficient here because it can't execute arbitrary data operations. But an agent with tool use can become a kind of *AI data engineer or data analyst* that interacts with data sources dynamically.

Consider a use case in business intelligence: *"Analyze our sales data for the last quarter, correlate it with marketing spend, and produce key insights and recommended actions."* This is a complex pipeline: one needs to query a sales database or CSV, possibly run statistical analysis or at least aggregate calculations, compare with marketing spend data (another source), then interpret the results and suggest actions. An

Agentic RAG system could approach this by using specialized tools at each step:

1. **Data Extraction:** The agent might use a SQL tool to query the sales database for Q4 data. It might also query a marketing database for the spend data per region or product.

2. **Data Analysis:** The agent could then invoke a Python execution tool to perform computations – e.g., compute correlations, growth rates, anomalies. (Alternatively, if the LLM is powerful enough, it might do some analysis in-text, but for reliable numeric analysis, calling a code tool is more accurate.)

3. **Insights Generation:** With the computed results (like "marketing spend up 20% led to sales up 30% in region A, but region B saw no growth despite spend"), the LLM agent can then generate insights: perhaps noticing patterns or outliers. The reasoning agent might even decide to drill deeper – e.g. if region B had no growth, the agent could query customer feedback data for region B (another retrieval) to find potential reasons. This is dynamic and not pre-programmed; the agent is following the trail of analysis as a human analyst would.

4. **Reporting:** Finally, the agent compiles the insights and recommended actions (maybe "reduce spend in region B or investigate product fit, increase spend in region A next quarter as it showed positive ROI"). This could be output as text, or the agent could populate a report template or even generate charts (if it has a tool to create simple charts or to call an API like a

plotting service).

This orchestration is clearly beyond a single QA. It's like a mini workflow automation with intelligence. A traditional data pipeline might require a human or a pre-defined script to do each part. An Agentic RAG agent can flexibly navigate the pipeline, perhaps even adjusting what data to pull based on intermediate findings (e.g., "sales are down unexpectedly in one category, let me pull inventory data to see if stock-outs were an issue").

Indeed, one of the *Agentic RAG system types* that Analytics Vidhya describes is **"Adaptive RAG"**, where the system adapts retrieval and processing based on contextanalyticsvidhya.comanalyticsvidhya.com. Another is **"Self-Reflective RAG"**, where the agent reflects on whether its answer might be wrong and corrects itselfanalyticsvidhya.com. In a data pipeline context, this could mean the agent checks if the analysis results make sense and if not, it might fetch additional data.

Agentic RAG's capacity to integrate structured data sources is key. In the survey, one example (though in the context of financial analysis, which overlaps with data pipelines) shows a *Hierarchical Agentic Workflow* where a top-tier agent prioritizes data sources, a mid-tier agent retrieves real-time market data via APIs and databases, and lower-level agents do web searches for news, finally aggregating the resultsarxiv.orgarxiv.org. This is essentially orchestrating multiple data retrieval actions in a pipeline to answer a query about investment options, and it is very analogous to orchestrating enterprise data sources.

Another concrete scenario: **ETL (Extract-Transform-Load) automation**. Suppose an agent is tasked with: “Every day, gather data from source X and Y, integrate them, and flag any anomalies or send a summary to the team.” Using Agentic RAG, the agent can each day: extract data from an API (tool call), extract from a database (another tool), transform or merge them (maybe using an internal tool or asking the LLM to format them consistently), then analyze (LLM or code tool for anomaly detection), then generate a summary (“today’s key metrics: ..., anomalies: ...”). It might even decide whether to send an alert email via an email-sending tool if anomalies are significant. This is like a smart autonomous data pipeline with decision points.

Enterprise workflow automation frequently involves connecting disparate systems – something agent tools can do on-the-fly. Microsoft’s Semantic Kernel has been positioned exactly for this kind of use: it calls them “goal-oriented applications” using agentic AI to manage long transactions across different APIs and endpointsinfoworld.com. Imagine an IT ops scenario: an agent monitors logs (via a tool that reads log data), detects an issue, queries a knowledge base for known solutions, and then either applies a fix (via calling a script) or escalates with a summary. This goes beyond RAG into automated operations, but RAG would be part of it (for querying knowledge bases of known issues or documentation). In fact, Moveworks (an enterprise AI platform) notes that agentic RAG can **automate routine tasks** and streamline operations by integrating with various enterprise resourcesmoveworks.commoveworks.com.

In summary, Agentic RAG can function as the intelligent glue in data pipelines, automating data retrieval and analysis steps that normally require static code or human effort. It brings a level of **dynamic decision-making** to workflows: instead of a fixed script, you have an agent that can alter its steps based on data. This is valuable for business analytics, operations monitoring, and any case where data from multiple sources must be combined and interpreted. It effectively democratizes a bit of the data analyst role to an AI: given a high-level goal, the agent figures out the technical steps to accomplish it and carries them out. This use case overlaps with the emerging idea of "AI assistants for business intelligence" or AI-driven analytics.

Enterprise Knowledge Management

Enterprise environments often have knowledge scattered across many systems – wikis, document repositories, emails, support tickets, databases, etc. Agentic RAG is especially useful for **enterprise knowledge management** because it can interface with multiple internal data sources and provide contextual, up-to-date answers to employees or customers. It acts as a smart company-wide assistant, breaking down silos of information.

Take the scenario of an employee asking: *"Has our company issued any updated policy on remote work benefits, and what are the key points?"* The relevant information might lie in a recent HR email, a PDF policy document on the intranet, and maybe some discussion notes. A traditional chatbot connected only to a Confluence wiki (for example) might miss it if the wiki isn't updated. An Agentic RAG agent could handle this by:

1. Checking the **enterprise wiki or knowledge base** via vector search for “remote work benefits policy”.

2. If it doesn’t find an update there, the agent might then search the **email knowledge base** (if it has an email tool) for company-wide emails about “remote work policy” in the last few months.

3. It could also search a **policies SharePoint or cloud drive** for any PDF or document named “Remote Work Policy 2025” etc.

4. After retrieving possibly an email announcement and the PDF text, the LLM part summarizes the key benefit changes and outputs the answer to the user. It might cite that it’s based on an email from Oct 2024 and a policy document. If integrated in an internal chatbot, it could even provide links to those sources if allowed.

The value here is that the employee gets a direct, synthesized answer that pulls from various company data, rather than them having to search each system individually. Glean (an enterprise search company) writes that Agentic RAG systems are especially useful when knowledge is fragmented across systems in an organizationglean.comglean.com. The agent acts as a *unified interface* to all knowledge: it decides which system to query and aggregates the information.

Another example: **customer support agent** (an external-facing knowledge bot). Many companies have loads of support tickets, FAQ documents, product manuals, etc. If a customer asks a complex question, the answer might require information from multiple places. An Agentic RAG support bot might:

- Search the FAQ database for a relevant answer (retriever 1),
- If not found, search the internal knowledge base or past ticket resolutions (retriever 2),
- Possibly run a diagnostic tool if it's a troubleshooting question (like querying a status API or knowledge graph of dependencies),
- Then present the answer or next steps to the customer.

This dynamic retrieval is noted to improve customer support automationmoveworks.commoveworks.com. Moveworks, in their blog, explicitly lists "Automated employee and customer support" as a prime use case for agentic RAG, noting it provides quick and precise answers and reduces workload on human agentsmoveworks.commoveworks.com. The agent can handle layered tasks – e.g., understanding the user's issue, gathering info from various internal tools (knowledge base, ticket history, etc.), and delivering a solution or helpful response. Essentially, it's an AI tier-1 support that knows when to escalate if needed.

Internal knowledge management also includes helping employees make informed decisions by querying data. We saw part of that in the data pipeline scenario. Even for more text-based knowledge, an agent can ensure no crucial info is missed by checking multiple sources. Another scenario: a new employee asks an internal chatbot: "How do I set up

VPN access and what's the policy on using personal devices for work email?" This touches IT documentation (VPN setup) and HR policy (BYOD for email). The agent can fetch from an IT wiki and an HR policy doc and compose a single answer. Traditional bots often fail when a question spans departments or knowledge bases; an agentic bot can route the query to multiple sources (like a *router agent* concept, deciding the query is multi-faceted)weaviate.iomedium.com.

Moreover, agentic systems can maintain context across a conversation, which is common in enterprise Q&A. An employee might ask follow-ups like "What about for contractors?" and the agent should remember the context (we're talking about device policy) and fetch that info. Memory and planning enable that continuation gracefully.

In essence, **Agentic RAG enhances enterprise knowledge management by being a *contextual meta-searcher and aggregator* across the organization's information landscape**. It not only finds information but understands how to combine it. This leads to more accurate and comprehensive answers. As an example, IBM's watsonx Assistant could leverage agentic RAG to integrate with various enterprise data sources (the survey mentions IBM's model answering complex queries by integrating external information)arxiv.org. Likewise, the Moveworks AI Assistant case study (Section 7) demonstrates using RAG with an agent to provide precise info access and automate info retrieval in an enterprise settingmoveworks.commoveworks.com.

Overall, companies implementing agentic RAG report improved **information discovery** and **decision support** internally. By ensuring the AI assistant can reach into all necessary nooks of data and reason about it, employees can trust they're getting an informed answer. This goes a long way in making enterprise AI assistants actually useful, moving beyond the often siloed and limited chatbots of the past.

Financial Analysis

The financial domain requires dealing with real-time data, numeric calculations, and multi-source information – all tasks well suited for Agentic RAG's tool-using, iterative approach. **Financial analysis assistants** can utilize agentic RAG to support analysts or even automate certain analytical tasks.

One compelling example (taken from the survey paper) is a **Financial Analysis Agent** answering a question like: "*What are the best investment options given the current market trends in renewable energy?"*arxiv.org. This is a complex question that requires:

- Knowledge of current market data (stock or asset performances in renewable energy sector),
- Knowledge of trends (possibly needing news about policy or technology in that sector),
- The ability to combine quantitative data with qualitative insights.

The Agentic RAG solution as described in the survey proceeds as a hierarchical multi-agent workflowarxiv.orgarxiv.org:

1. A **Top-Tier Agent** first analyzes the query's complexity and decides which sources to prioritize. For investment advice, it might prioritize reliable financial databases and economic

indicators over, say, random social mediaarxiv.org. This is an important step – filtering out noise and planning which info is needed (e.g., "need real-time market data and recent policy news").

2. A **Mid-Level Agent** then retrieves *real-time market data* such as stock prices, sector indices, etc., likely via financial APIs or querying a database of market dataarxiv.org. It could also gather structured data like P/E ratios, growth rates, etc., for top companies in renewable energy.

3. **Lower-Level Agent(s)** perform web searches for recent news or policy announcements regarding renewable energy (maybe something like "new subsidies for solar in EU" or "international climate agreement impact on renewables")arxiv.orgarxiv.org. They might also consult recommendation systems or analyst reports (the example mentions tracking expert opinions and news analytics)arxiv.orgarxiv.org.

4. **Aggregation and Synthesis:** The results from data and news are then compiled. The agent integrates quantitative data (market growth, stock performance) with qualitative insights (policy support, expert opinions)arxiv.org. The final answer might be something like: "*Renewable energy stocks have seen 15% growth this quarter driven by supportive government policies; wind and solar are likely to continue momentum, though emerging tech like green hydrogen is higher risk/higher reward.*"arxiv.org. Indeed, the survey's example response is along those linesarxiv.org.

This case shows multiple agentic features: multi-step retrieval (data + news), using tools (financial API, web search), reasoning to combine them, and even specialization of agents. A traditional system might give a generic answer or only one facet (e.g., just pulling some static report). The agentic approach yields a more **comprehensive and up-to-date analysis**arxiv.org.

Other financial use cases:

- **Portfolio Q&A:** An investor could ask, "How did my portfolio perform compared to the S&P 500, and what contributed to the difference?" An agent could retrieve the user's portfolio data (perhaps via a database or API), fetch S&P 500 data, compute performance metrics, and then identify contributing factors (maybe via news: e.g., "Stock X in your portfolio dropped due to an earnings misswandb.ai"). It could even cite relevant news for those specific stockswandb.ai.

- **Market Surveillance:** Agents could monitor multiple sources – news, social media, market data – to flag anomalies or opportunities. For instance, if an agent sees unusual trading volume in a stock plus some breaking news in that sector, it could highlight that to a trader. This is an extension of use of RAG in finance where LLMs read news for sentiment; here the agent can also cross-check data, possibly even run risk models (via tool).

- **Financial Document Processing:** Agentic RAG can help parse and analyze financial filings or reports. A prompt on a conference call transcript: "Summarize the key risks mentioned by the CFO and check if any metrics were unusual.” The agent can scan the transcript (retrieval), identify risk statements, maybe compare stated metrics to previous quarter (another retrieval or DB query), and output a summary with any warnings. Actually, there's a Medium article titled “Revolutionizing Financial Document Processing with Agentic RAG”medium.com, which likely covers an example like reading a PDF filing and extracting specific analysis – an agent could use a PDF reading tool and an analysis script to do so.

Financial analysis often involves **numbers and facts that must be precise**, which is why tool use (like calculators or code) is important – to avoid the LLM making arithmetic mistakes. Agentic RAG allows plugging in those calculation tools. It can also interface with existing financial models or libraries. For example, an agent could call an internal VaR (Value at Risk) calculation function on a portfolio data set and then have the LLM explain the result in plain English.

Another useful agentic feature in finance is **alerting and decision-making**. Because finance is time-sensitive, an agent might have a rule to not just answer questions but to proactively monitor triggers. While that ventures into agentic behavior beyond user-posed queries, one can imagine a scenario where an agent decides to get more data if something looks off. For instance, if a metric is outside expected range, it might dig deeper on its own, something a static system wouldn’t do.

The benefit of Agentic RAG in finance is well captured by IBM’s point on **scalability and adaptability**: with networks of RAG agents tapping into multiple data sources and using planning, the system can handle a wide range of queries and tasksibm.com. In finance, where questions can range from simple ("what's the price of X?") to very complex ("analyze the impact of recent Fed announcements on my bond holdings"), having that adaptable workflow is crucial. Traditional RAG would be limited to maybe retrieving a snippet from a financial article or a single number.

To ensure accuracy and compliance (given finance is a regulated area), such agents would be used as decision support for humans rather than fully autonomous decision-makers, at least initially. But even as support, they can greatly speed up analysis that requires sifting through Bloomberg terminals, financial statements, and news – tasks that human analysts do daily.

Legal Summarization and Analysis

Legal work involves heavy documents, precise language, and connecting various sources (e.g., case law, statutes, contracts). Agentic RAG can act as a legal assistant to summarize and analyze legal documents while cross-referencing relevant laws or prior cases – something a simple RAG QA might not fully accomplish.

A prime use case is **contract analysis and compliance checking**. Suppose a company's legal team wants to review a vendor contract and identify any clauses that might violate company policy or regulatory requirements. An Agentic RAG system could:

1. Use a **document reader tool** to parse the contract (if it's a PDF or scanned, possibly using OCR).

2. Use the LLM to extract key clauses or obligations from the contract text.

3. For each key clause, use a **knowledge base search** to find any internal policies or legal guidelines that relate (e.g., the company's compliance rules stored in a policy database, or relevant laws).

4. If conflicts are found, the agent can highlight them and explain – e.g., "Clause 5 allows data sharing with third parties, which conflicts with our internal data protection policy Section 3.2" (citing the internal policy).

5. Compile a summary of the contract with flagged issues and maybe suggest revisions.

This process is exactly what a human lawyer or compliance officer might do: read contract, recall or look up rules, compare and flag. Agentic RAG can assist by doing the lookup and initial comparison automatically. Glean's example earlier about summarizing a contract and flagging procurement policy violations is a direct illustrationglean.comglean.com. The agent specialized: one agent extracted key clauses, another retrieved internal policies, a third compared and highlighted conflictsglean.comglean.com. Because each agent can focus (document parsing vs policy retrieval vs comparison), it's efficient and modular.

Beyond contracts, **legal research** is a big domain. Lawyers often need to find relevant case law given a situation. An agentic RAG can perform multi-hop research: start with a query about a legal question, search a case law database (like LexisNexis or others) for relevant cases, then possibly read a found case summary and extract principles, then search again for any appellate history, etc. It can then provide a synthesized answer like, "Under [Case Name, Year], the courts held X, which suggests in our scenario Y would apply." Traditional RAG could search and maybe retrieve a snippet from one case, but an agent can verify if multiple cases agree, or if it needs to search statutes as well.

Another use is **summarizing deposition transcripts or discovery documents**. Those can be hundreds of pages. An agent could chunk through them, each time retrieving parts and summarizing, and then cumulatively build a summary or a timeline of events. If it finds a particular item of interest (say mention of a key event), it could search correspondence or emails for corroboration (like linking evidence across document types). This is hugely time-saving in legal discovery, where teams comb through piles of documents for relevant info.

Legal writing often demands **precedent and citations**. An agentic system could be prompted: "Draft a brief arguing X, citing relevant precedents." The agent would find relevant cases (retriever), perhaps use a tool to Bluebook format the citations, and then have the LLM draft the argument using those citations. While this is advanced and would need heavy verification by lawyers, it can serve as a first draft generator, drastically reducing drafting time.

A notable challenge in legal AI is ensuring no hallucinated case law – making up a case would be disastrous. Agentic RAG's approach of retrieving actual case text before the LLM cites it helps avoid that (the LLM isn't just guessing a case, it's quoting from actual retrieved cases). The agent's self-checking nature can also verify that each cited case is relevant and not contradicting the argument. We already see glimpses of this: some products use LLMs to read and annotate contracts, but an agentic approach would be more robust by explicitly checking compliance.

The arXiv survey mentions legal applications and highlights combining semantic search with legal knowledge graphs to automate contract review, ensuring compliance and mitigating risksarxiv.org. This essentially describes the contract analysis use case – using structured knowledge (like a knowledge graph of legal rules) plus search to vet a contract. The agent could query a knowledge graph of regulations to see if any clause might trigger a compliance issue. For example, a clause about storing customer data in another country might be cross-checked with GDPR rules via a knowledge graph query – a complex multi-source check.

Additionally, **court docket management** might use agentic RAG: an agent that reads new filings and alerts attorneys to any deadlines or requirements, pulling out key info and linking to relevant rules of procedure.

In summary, legal summarization/analysis with Agentic RAG can:

- **Summarize and cross-check** large legal documents (contracts, legislation, case transcripts).

- **Answer legal questions with multi-step reasoning** (finding and comparing multiple sources of law).

- **Ensure compliance** by comparing documents to a set of rules or policies.

- **Draft and cite** documents by pulling in the appropriate references.

The use of tools is crucial: integration with databases of cases, ability to use custom functions (e.g., a function that checks if a clause exists in a contract template library), and memory to handle context of a case (facts of a case stored and reused when analyzing

5. Limitations and Where Agentic RAG Fails

While Agentic RAG is powerful, it is not without pitfalls. Understanding its limitations is important for setting correct expectations and identifying areas for improvement. Key failure modes and challenges include: **tool execution errors**, **hallucination or error propagation**, **increased latency and cost**, and **retrieval bottlenecks or blind spots**, among others.

- **Tool Execution Errors & Complexity:** Incorporating external tools introduces new points of failure. If an agent formulates a tool call incorrectly (e.g., malformatted API query or code), the tool might throw an error or return nonsense. The agent must then handle this – which not all implementations do gracefully. Poor error handling can cause the agent to get stuck in loops or fail silently. A comparative analysis of agent frameworks noted, for instance, that without careful prompt design an agent might loop infinitely or misuse tooltheflyingbirds.in】. Coordinating multiple tools and agents also adds complexity: as the survey succinctly puts it, “managing interactions between agents requires sophisticated orchestrationarxiv.org】. An Agentic RAG system might involve many moving parts (multiple agents, each with several tools); orchestrating this without bugs is non-trivial. This complexity can lead to integration errors or edge cases where the agent’s plan goes awry (for example, issuing a web search for an internal database query due to a misunderstanding). Developing and testing agent workflows is thus more complex than a static pipeline – there is an inherent **coordination complexity** in multi-agent or multi-tool setuparxiv.org】.

- **Hallucination Propagation:** One goal of RAG is to reduce hallucinations by grounding the LLM in retrieved data. Agentic RAG, by virtue of iterative retrieval and checking, often succeeds in this – but not always. If the needed information is not present in any accessible source, the agent could still end up fabricating an answer after exhausting its tools. Worse, if the agent retrieves misleading or irrelevant information and doesn’t detect its irrelevance, the LLM may incorporate it into its reasoning, essentially **propagating errors**. A subtle issue is that the agent’s chain-of-thought (which is hidden from the end-user) might contain hallucinated rationale that influences its actions. For example, an agent might incorrectly “think” an obscure law is relevant and then spend steps trying to find it. Traditional RAG would have simply said “no info found” if nothing was retrieved, but an agent might generate a mistaken intermediate conclusion that sends it on a wild goose chase. In effect, an agent can **hallucinate the importance of irrelevant data** and waste cycles or produce a confusing answer. That said, many agent frameworks try to mitigate this: e.g., by having a **Relevance Evaluation Agent** validate retrieved documentarxiv.org】 or by prompting the LLM to not assume facts not in evidence. Nonetheless, hallucinations are not eliminated – they can recur if the retrieval fails to find truth. As OpenAI's function-calling showed, an LLM will follow a tool-based approach but if tools return nothing useful, the model might still fill the gaps with guesswork. Therefore, Agentic RAG reduces hallucinations **on correctly retrieved information*weaviate.io】, but if that information is wrong or insufficient, the final output can still be wrong, just with more steps involved.

- **Latency and Throughput:** Agentic RAG systems are slower and more resource-intensive than straightforward QA systems. Each additional tool invocation or reasoning step accumulates latency. An agent might perform several sequential searches and LLM calls before producing an answer. For user-facing applications, this can impact experience – users may have to wait several seconds longer for responses. In high-throughput scenarios (like handling thousands of queries per second in a customer service bot), an agentic approach could become a throughput bottleneck. The survey notes that "the dynamic nature of the system can strain computational resources for high query volumesarxiv.org】 and identifies **scalability and latency issues** as a challenge for traditional RAG that agentic systems must also grapple witarxiv.orgarxiv.org】. Essentially, agent loops that are fine for one-off complex questions might not scale linearly when many questions are asked. Caching and parallelization can help (e.g., do web search and DB query in parallel if possible), but the complexity can sometimes serialize actions (an agent often waits for one tool to finish before deciding the next step). Moreover, each LLM invocation costs tokens/money. Running an agent for a single query could consume several times the tokens of a single-turn answer, which directly translates to higher API costs or more GPU time. Thus, **cost-effectiveness** is a concern – organizations must weigh the improved capabilities against the increased compute expense. In practice, many deployments use a hybrid approach: use cheaper methods for simple queries and reserve agentic workflows for the hard cases.

- **Retrieval Bottleneck and Knowledge Gaps:** Agentic RAG is only as good as the tools and data it has access to. If some knowledge is not in any database or the agent doesn't have a tool to get it, the system may still fail to provide a correct answer. For example, if an organization's data is not indexed in the vector store, the agent can try web search or other APIs, but ultimately it cannot retrieve what isn't there. This is essentially the same limitation as RAG (need a comprehensive knowledge source), just spread over multiple sources. There is also a **retrieval bottleneck** in terms of quality: if the search or retrieval component yields poor results (say the vector search isn't tuned well and misses relevant documents), the agent's reasoning won't magically fix that – it might even be led astray by whatever was retrieved. A known limitation of RAG is that it struggles with queries requiring synthesis of info that is never explicitly stated in any single document; an agent might try multiple searches and still not find an explicit answer, and at best it could attempt an inferred answer – which could be a hallucination. Essentially, the agent cannot truly "create" new knowledge; it can only remix what it finds or logic it out with general knowledge. If both fail, it fails. We might call this the **coverage limitation** – if the union of all accessible sources doesn't contain the answer, an Agentic RAG will eventually hit a wall (though it may not always admit it; it could give the best guess).

- **Error Propagation in Multi-step Reasoning:** In multi-step reasoning, an early mistake can compound through later steps. For example, if an agent initially misunderstands a user query and picks a wrong retrieval topic, all subsequent steps might be operating on a wrong premise. Traditional RAG would have just given an irrelevant answer or "no answer found." An agentic system might seem to confidently go through several steps, giving the impression of thoroughness, but ultimately end up with an irrelevant or incorrect answer because the initial assumption was wrong. This can make debugging tricky – which step did the agent go wrong on? The opacity of LLM reasoning (even if we have the chain-of-thought, it can be hard to interpret) complicates validation. It's a known challenge to make these systems **interpretable and debuggable**. Researchers note the lack of specialized evaluation metrics – how do you score if an agent took unnecessary steps or used a suboptimal tool, as long as the final answer is right? The development of benchmarks capturing agentic subtasks is still ongoinarxiv.orgarxiv.org】.

- **Safety and Containment:** Giving an agent the ability to act (especially if it can perform writes or make changes via tools) raises safety concerns. A misbehaving or vulnerable agent could, in theory, execute harmful actions (for example, an agent plugged into a shell tool could execute destructive shell commands if prompted maliciously). In enterprise settings, tools are usually constrained (read-only or with limited scope), but it's a consideration. There's also the risk of exposing sensitive data: an agent might pull internal documents to answer one user's query, and then if asked a question by another user, mistakenly include content from the previous query due to long-term memory misuse. Proper **session isolation** and permissioning of data are important to prevent such leaks. These are more operational concerns than fundamental limitations, but they're failures that can occur if the system is not carefully designed. As IBM's commentary suggests, ensuring *ethical decision-making* and *responsible deployment* is an active challengarxiv.orgarxiv.org】. For instance, an agent might need to be prevented from giving certain advice (legal, medical, etc.) or from disclosing private information even if it has access to it. Implementing these constraints in an agent that is designed to be autonomous is an ongoing area of work.

- **Prompt Fragility:** *Agentic systems often rely on complex prompts* to guide the LLM's behavior (the planner prompt, tool usage instructions, etc.). These prompts can be fragile – a slight change in wording or an unexpected user query format might throw off the agent's guidance. There is a risk that the agent might interpret the instructions in a way not intended by the designers, leading to mistakes. While this is also true in simpler LLM setups, the more elaborate the prompt (especially with few-shot examples of how to use tools), the larger the prompt and the greater chance of some token causing a distributional shift in the model's output. Some frameworks use structured policies or even code to enforce agent behavior (e.g., Microsoft's Semantic Kernel Agent Framework in .NET uses code to manage steps, reducing reliance on prompt aloninfoworld.com】), which can mitigate prompt-based brittleness. But many current agents (especially those implemented purely via LangChain-style prompts) can occasionally glitch – e.g., output an action in the wrong format, or start role-playing outside the intended role if the conversation takes a weird turn. These are usually edge cases, but they highlight that we're still in relatively early days of robust agent design.

In summary, Agentic RAG can fail in both mundane and surprising ways. It might simply be slower and heavier than desired, or it might chase a wrong hypothesis down a rabbit hole. It might hit tool or data limitations, or produce a very authoritative-sounding answer that is subtly off because of a mistake five steps earlier. Many of these limitations are active research areas, as we will discuss in Future Directions. As practitioners, being aware of these failure modes allows us to introduce mitigations: for instance, fallback to a human or a simpler system if the agent loop exceeds a certain number of steps (to prevent infinite loops), logging and tracing agent decisions for debugginlangfuse.comlangfuse.com】, sandboxing tool execution, and thorough testing on known queries to see where the agent might stray.

6. Key Tools, Frameworks, and Libraries for Agentic RAG

Implementing an Agentic RAG system from scratch is a complex endeavor. Thankfully, a variety of frameworks and libraries have emerged to provide building blocks for agents, tool integration, planning, and memory. Here we compare some of the prominent ones: **LangChain (Agents)**, **CrewAI**, **AutoGen**, **Semantic Kernel**, and **DSPy**. Each takes a slightly different approach to constructing agentic workflows. We will look at their core philosophies, strengths, and typical use cases.

LangChain Agents (Python)

LangChain is one of the earliest and most popular frameworks for developing applications with LLMs, especially for RAG and agent scenarios. LangChain provides a modular set of components to handle prompts, memory, retrieval, and integration with a vast ecosystem of tools (APIs, databases, etc.arxiv.org】. LangChain introduced the concept of **Agents** following the ReAct paradigm – basically, it can wrap an LLM with logic to decide which "tool" to use next, given the LLM's output.

- *Philosophy & Approach:* LangChain is very *prompt-centric*. Agents in LangChain are often configured by providing the LLM with a prompt template that lists available tools (with usage instructions) and perhaps an example of the thought-action-observation sequence. LangChain's design emphasizes chaining multiple steps or sub-calls; an agent is essentially a loop that keeps querying the LLM for an "action" until it outputs a final answer. It supports various agent types (zero-shot, ReAct, self-ask, etc.) to accommodate different styles of reasoning. The core concept is the **Chain** (hence

LangChain) – a sequence of actions or calls can be strung together, sometimes dynamically by the agent.

- *Integrations:* One of LangChain's biggest strengths is its large community and integration catalotechtarget.com】. It has out-of-the-box tools for web search, SQL, Python execution, Wikipedia, WolframAlpha, and many more. It also integrates with vector databases (Pinecone, Weaviate, etc.) for retrieval, and with model providers (OpenAI, HuggingFace, etc.). This means a developer can easily add a new tool by using a LangChain wrapper, and the agent can call it. However, as TechTarget notes, the vast majority of LangChain's integrations are community-contributed and may vary in quality or maintenanctechtarget.comtechtarget.com】.

- *Strengths:* Rapid development and prototyping of agentic apps. Because LangChain handles the prompt wiring, tool invocation logic, and includes memory modules, a developer can focus on what they want the agent to do. LangChain's **LLM abstractions** allow switching between models easily, and its memory abstractions (short-term buffer, long-term vector memory) simplify adding context. It's pure Python (with some JS and other language variants available), making it accessible. LangChain also now offers a "graph" execution (LangChain Hub / LangFlow and LangGraph) for more complex control flows rather than linear chainlangfuse.com】.

- *Weaknesses:* The flexibility can be a double-edged sword. Because it relies on prompting the LLM to follow the protocol, agents can sometimes break protocol (especially with newer models that might not exactly follow the expected output format). There have been reports of LangChain agents being hard to debug, as the chain-of-thought (if not printed out) is hidden. Performance-wise, LangChain doesn't inherently optimize multiple tool calls (though you can manually parallelize certain things). Also, the heavy reliance on OpenAI API and similar means using it at scale will incur costs (not a LangChain-specific flaw, but a reality of these agent calls). Compared to more "structured" frameworks (like Semantic Kernel's planner), LangChain's approach might feel a bit unstructured – it's basically prompt engineering under the hood. Nonetheless, LangChain has matured significantly and remains *the go-to framework for many experimenters in the agentic space*, especially for Python users.

Use Cases: LangChain agents have been used in numerous hackathon projects and prototypes: from "ChatGPT with internet access" style chatbots, to workflow assistants. For instance, people have built travel planners (the agent queries flight and hotel APIs), personal assistants (managing calendars, to-dos via tools), and research bots (as described earlier). The large set of tools makes it a generalist solution. Enterprises sometimes start with LangChain for POCs, but some eventually migrate to more controlled frameworks for production. Still, even companies like Microsoft referenced it – e.g., the LangChain integration with LlamaIndex's LangGraph for more complex graphs in RAarxiv.org】.

CrewAI (Python)

CrewAI is a newer framework that emphasizes multi-agent collaboration and high performance. It brands itself as a lean, fast alternative to LangChain, built from scratch without the latter's dependenciedocs.crewai.com】. The name "CrewAI" hints at a team of AI agents working together (a "crew" of agents). It provides an infrastructure for defining multiple agents with roles and orchestrating their interactions.

- *Philosophy & Architecture:* CrewAI's core concept is around **role-based agent teams** and a manager that coordinates thetheflyingbirds.in】. It has two key abstractions: **Crews** and **Flows*docs.crewai.com】. *A Crew* is like a container for a set of agents with specific roles working towards a common goadocs.crewai.com】. *A Flow* is a workflow (potentially event-driven or one-off) that can involve single or multiple LLM calls and integrates with the crew. CrewAI encourages designing agents similar to a human organization: for example, you might have a "Researcher" agent and a "Writer" agent collaborating (like we described in the customer support analysis case study earlier, where one agent analyzes data and another writes content). CrewAI provides infrastructure for these agents to delegate tasks to each other and communicate via a shared memory or messaging.

- *Integrations & Tools:* CrewAI comes with a suite of built-in tools (the documentation lists tools for web searching, various RAG (Retrieval) tools for different vector DBs, scraping, code execution, etc.docs.crewai.comdocs.crewai.com】. It also integrates with model providers (OpenAI, etc.) but importantly is model-agnostic. It doesn't piggy-back on LangChain's integrations; it has its own implementations, which the creators claim are more optimized. A glance at CrewAI's docs shows specialized RAG tools for different DBs like Weaviate, Qdrant, Pinecone, plus others for scraping websites, reading PDFs, SQL, and mordocs.crewai.comdocs.crewai.com】. This suggests a focus on enterprise data integration.

- *Strengths:* CrewAI is designed for **performance and scalability**. It aims to reduce overhead – for example, its creators highlight it's *"lightning-fast… independent of LangChain"* which implies they optimized how agents call LLMs and handle memordocs.crewai.com】. It also provides a structure (manager agent and crew) that helps avoid chaotic agent interactions – more controlled than just prompting an LLM to call other LLMs arbitrarily. The role specialization is a logical way to break down tasks. This can result in better organized prompts (each agent can have a prompt tailored to its role). CrewAI also supports YAML-based configuration of agents, which can help in managing complex setups and making them more deterministic (rather than purely relying on learned behaviortheflyingbirds.intheflyingbirds.in】. Another key feature is *structured output*: it can enforce that agents produce output in JSON or a schema (via Pydantictheflyingbirds.intheflyingbirds.in】, which is useful for ensuring the final output or intermediate communications are parseable and follow a contract (important in production settings).

- *Weaknesses:* As a newer framework, CrewAI has a smaller community and ecosystem compared to LangChain. This means fewer example pipelines publicly available, and possibly limited community-contributed tools (though it has many out-of-box, a developer might need to implement any custom integration that's not already supported). The learning curve might be moderate – understanding the Crew/Flow abstractions and how to design multi-agent workflows effectively requires a bit of conceptual work (though their analogy to departments and team members helpsdocs.crewai.comdocs.crewai.com】. Also, while performance is a goal, ultimately the underlying LLM calls still dominate latency; CrewAI can optimize the orchestration overhead but cannot speed up the model itself (beyond maybe supporting async parallel calls in flows, which is valuable).

Use Cases: CrewAI shines in scenarios where you want *multiple agents working in a coordinated way*. For example, building an "AI team" for a complex task: one could envision a **project management AI** where one agent breaks a project into tasks, then assigns each task to specialist agents (say, coding agent, testing agent, documentation agent), then integrates results – CrewAI's design is explicitly intended for that kind of pattern (with a manager agent delegatingdocs.crewai.comdocs.crewai.com】. Another use case is any multi-step process that benefits from different skill sets: e.g., a legal analysis agent (role: finds relevant laws) plus a reasoning agent (role: apply them to a scenario) plus a writer agent (role: draft a memo). Because CrewAI offers granular control, it's likely used in enterprise experiments where reliability is key – e.g., a financial report generator with separate agents for data retrieval and language generation, to ensure the numeric analysis and the narrative are handled properly and can be validated separately. The framework's emphasis on being *enterprise-ready* (they mention many developers certified and a trend towards standardizatiodocs.crewai.com】) suggests it's aiming for production use in organizations that might have found LangChain too slow or unpredictable.

AutoGen (Microsoft)

AutoGen is an open-source framework from Microsoft Research that focuses on enabling *multi-agent conversations*, particularly between LLM agents and tool-using agents, including the concept of agents that can converse with each other to solve taskmicrosoft.commicrosoft.com】. It arose from research on letting multiple LLMs collaborate (for example, a "user" agent and an "assistant" agent chatting to refine a solution, or an "analyst" and a "coder" agent pair working together). AutoGen (sometimes stylized as "AG2" in its latest versioarxiv.org】) is behind some of the academic benchmarks and examples of agents talking to agents.

- *Philosophy:* AutoGen's key idea is to *compose multiple agents that converse to accomplish tasksmicrosoft.com】. It treats each agent as having a persona and goal, and they communicate via a conversation interface. This is different from the single-LLM-with-tools model; instead, you might have one agent that's good at one thing and another agent good at another, and they talk in natural language (or a structured message protocol). The framework provides the scaffolding to create these agents (with possibly different LLM backends or the same LLM with different system prompts) and a channel for their messages. It also integrates tool use by allowing agents to be associated with tools.

- *Features:* AutoGen agents are *customizable and conversable*, and the framework supports various modes (LLM-only, LLM + human, LLM + tools, or multi-LLMmicrosoft.com】. A highlight is that it allows **human involvement** as an agent too – e.g., you can easily loop in a human as one of the agents in the conversation, which is useful for human-in-the-loop use cases. It also enables developers to define conversation patterns or protocols. For instance, one can set up an architecture where Agent A must get approval from Agent B (or a human) before finalizing. AutoGen also provides mechanisms to manage the conversation history and state across these agents.

- *Strengths:* AutoGen excels in scenarios requiring collaboration or complex negotiation between different competences. A classic example from MSR is pairing a reasoning agent with a code-generation agent: the reasoning agent might break down a task and the coding agent writes code for parts of it, then the reasoning agent tests or critiques the code – this was demonstrated as a way to have an "AI Pair Programmer" that self-checkmicrosoft.commicrosoft.com】. In such tasks, the conversational style helps because one agent can naturally ask the other for clarification or more info ("Can you write a function to do X?" "Here it is." "I see an error Y, can you fix that?" etc.). AutoGen is built as a *generic infrastructure* and has been used in pilot applications in domains like mathematics, coding, supply chain optimization, and decision-making gamemicrosoft.com】. The fact that it won a best paper (at the LLM Agents workshop) indicates its design was well-regarded academically.

- *Weaknesses:* Multi-LLM conversations can be token-expensive – effectively you have two or more models chattering, which can blow up context windows and usage quickly. Also, ensuring the conversation stays on track requires careful prompt setup; otherwise, agents might go in circles or agree on something incorrect. AutoGen's focus is on the agent communication layer; it still relies on underlying LLMs to be competent at their roles. If the models used are weak, the conversation won't magically produce good results. Another practical consideration: debugging a multi-agent convo can be even more complex than debugging a single-agent chain, because you have to trace which agent said what and why. However, one could argue the explicit messaging makes it a bit easier to follow logically than a single agent's hidden chain-of-thought.

AutoGen is quite powerful but may be best suited for tasks where *two (or a few) heads are better than one*. If a single model can do the job with tools, you might not need multiple agents. But if you want specialization (like one agent focused on correctness and another on creativity, or one teacher and one student model scenario), AutoGen is a great fit.

Use Cases: AutoGen has been used in:

- **Coding agents**: e.g., an agent that writes code and another that executes and debugs it – the MSR paper demonstrated that approach to get more reliable code generatiomicrosoft.com】.

- **Analyst + Assistant** patterns: one agent playing the role of a curious user asking questions, the other as an expert answering – forcing the system to articulate reasoning. This is similar to self-query or self-refinement approaches.

- **Workflow automation with guardrails**: one agent could be tasked with creativity and proposing solutions, and another agent could be more conservative and check proposals against constraints (e.g., one generates a marketing copy, another checks it for compliance with brand guidelines, then either approves or requests changes).

- Because AutoGen supports human agent, it's used in scenarios where a human might intermittently join the loop. For example, a customer support triage: an AI agent tries to handle the issue, and pings a human agent if it gets stuck – with AutoGen, both are just participants in the conversation flow.

AutoGen's development also influenced other frameworks; for instance, Microsoft's Semantic Kernel is integrating with AutoGen to leverage its agent communication for complex workflowinfoworld.com】.

Semantic Kernel (C# / .NET, with Python bindings)

Semantic Kernel (SK) is an open-source SDK from Microsoft that initially was aimed at orchestrating prompts and skills for LLMs, especially in .NET applications. Over time, it evolved to include an **Agent Framework** for building agentic AI with a focus on *goal-oriented workflows* and integration with enterprise systeminfoworld.com】. Semantic Kernel is notable for being strongly typed and structured (leveraging the static typing and IDE friendliness of C#) and for integrating deeply with Azure AI services.

- *Philosophy:* Semantic Kernel's approach is about **programmatically orchestrating AI** rather than doing it all in the prompt. It introduces the concept of **Plugins (Skills)** which are essentially the tools or functions the AI can call, and a **Planner** that can compose sequences of these skills to achieve a given goainfoworld.com】. The Agent Framework in SK is built as a set of .NET libraries that manage the conversation, tool invocation, and state, rather than relying purely on the LLM to decide the floinfoworld.com】. This gives the developer more control and the resulting agent behavior can be more predictable. Microsoft's vision is to enable "goal-oriented applications" where you give the system a high-level goal and it figures out how to use available functions to fulfill iinfoworld.com】.

- *Integration & Tools:* SK is tightly integrated with the Azure ecosystem – it easily connects with Azure OpenAI, and with Azure Functions or other APIs as skills. Out of the box, SK provides a range of useful plugins (e.g., for memory, for math, for web search via Bing, etc.) and you can also wrap any REST API or C# function as a skill. Because it runs in .NET, it can naturally integrate with enterprise databases, systems, and authentication mechanisms – appealing for enterprise architects. SK also has memory persistence (it can use vector stores to store embeddings of past interactions). An interesting feature is that SK's planner can use AI to determine which skills to use in what order for a goal (this is somewhat like an automated planner that generates a plan, as an alternative to an agent loop) – they had something called "Sequential Planner" which is more like PDDL-style planning using LLMs. However, the Agent Framework introduced later is more akin to agent loops with function calling, but with the scaffold in code.

- *Strengths:* **Reliability and Enterprise readiness**. Because SK agents are coded in a real programming language, you can implement logic around the AI. For example, you can explicitly catch when the AI requests to use a certain tool and inject approval steps or modify arguments. This structured approach can reduce the risk of prompt misinterpretation. Also, SK's design encourages separation of concerns: your skills (tools) are distinct from the orchestration, which is distinct from the model prompt. This modularity makes it maintainable for large projects. Another strength is **integration with AutoGen** (planned) and a common agent runtime that Microsoft is developininfoworld.com】 – this hints that SK will leverage best-of-breed techniques (like MSR's research) in a production-friendly package. SK also is optimized for **memory management** – it can decide what context to bring in or not, and because it's code-driven, you can implement complex memory retrieval logic (like only inject the most relevant summaries to the context to avoid context overflow). In contrast, something like LangChain typically just appends everything or uses relatively simple strategies by default.

- *Weaknesses:* SK was initially .NET-first, which meant non-.NET users had a barrier (though there is now a Python binding for SK, it's not as mature or idiomatic as the .NET version). The .NET focus is great for enterprise shops that use C#, but less so for the open-source community which is largely Python-oriented for ML/AI work. Documentation for SK agents is improving but is not as rich as LangChain's simply due to community size. Another potential weakness is that SK being more structured might have a steeper learning curve for those not familiar with the concepts – you have to understand the Planner, Orchestration, etc. (However, engineering managers might appreciate the explicitness). In terms of performance, SK's overhead is minimal (just function calls in a program), so performance largely depends on the model calls themselves. One consideration: SK encourages using Azure OpenAI, which might be the route for enterprise deployments, but those who want open-source models would need to put in some work (though SK can call local models via Python if set up).

Use Cases: Semantic Kernel is geared towards **enterprise AI orchestration**. Think of scenarios like:

- An enterprise virtual assistant that can handle an employee request like "I'm going on vacation for 5 days, set my OOO reply and delegate my critical alerts to John." This requires planning:

update calendar, set OOO in Outlook, maybe message John – multiple steps with different connectors. SK excels at that because each of those actions is a plugin and the planner can assemble them, and you can supervise it.

- **Complex multi-step workflows** that involve interacting with various enterprise systems (ITSM tools, databases, CRM) where reliability is key. SK's agent framework was highlighted in Microsoft Ignite 2024 with use cases in IT automation and business process automatioinfoworld.cominfoworld.com】.

- Building **chatbots with memory**: SK's memory plugins allow building a bot that remembers previous conversations or user preferences over long periods (semantic caching). Because SK can be connected to long-term storage or knowledge graphs, it can over time build up a knowledge base for the agent.

- Also, SK can be used just for RAG-style Q&A with added steps. For example, Moveworks (though they have their own platform) is the kind of use case SK targets: answering enterprise questions by querying multiple sources – SK could orchestrate the search in SharePoint, search in emails, then combine results.

In short, SK is a strong choice if you need *structured agentic behavior integrated into a larger software system*, particularly in the Microsoft/Azure context. It's less of a standalone agent playground (like LangChain is) and more of an SDK to embed agent capabilities into enterprise applications.

DSPy (Declarative Self-Improving Python)

DSPy is an emerging framework from the academic realm (developed with contributions from Stanford, etc.) that takes a different stance: *programming, not prompting, language modelsdspy.ai】. It allows developers to create AI programs using Python constructs, where prompts are under the hood. The idea is to provide higher-level abstractions (like *modules* for predict, retrieve, etc.) and use optimization techniques to automatically tune prompts or even fine-tune models for better performance.

- *Philosophy:* DSPy (Declarative Self-Improving Python) emphasizes a **declarative programming model for LLMs*dspy.aitheflyingbirds.in】. Instead of hand-crafting a prompt flow, you declare what you want to happen in code and DSPy figures out how to prompt the model to do it. It treats components like modular functions. For example, you might declare a `Retrieve` module with certain properties (like which corpus to retrieve from), a `ChainOfThought` module, etc., and then compose them. DSPy can then generate the necessary prompts or training to make the model execute those modules effectively. Importantly, DSPy includes **optimizers that can automatically tune prompts** and even adjust model weights via few-shot fine-tuning or calibratiodspy.ai】.

- *Features:* Out of the box, DSPy provides a library of prebuilt modules for patterns like ReAct, Tree of Thoughts, etc., and tools integration (like a Python execution tool, embedding models for retrievaldspy.ailearnbybuilding.ai】. It also offers an *optimization framework* to improve performance: for instance, given some evaluation criteria or examples, DSPy can refine the prompt of a module to reduce errors. This is quite unique; most frameworks assume the prompt is static. DSPy's approach is somewhat akin to how a compiler might optimize code – here it's optimizing prompt/program performance. The declarative style also means you can more easily reason about the *flow* of data (and thus ensure determinism where needed). DSPy also emphasizes **model-agnosticism and generalization**: you can write a DSPy program and run it on different LLMs, and it tries to ensure it works similarly (with some behind-the-scenes tuningtheflyingbirds.intheflyingbirds.in】.

- *Strengths:* The biggest strength is **quality and reliability of output** due to prompt optimization. For tasks that require high accuracy (like factual correctness, following a format strictly), DSPy's ability to fine-tune or auto-adjust prompt weights is powerful. In a way, it blends the boundaries between using an LLM as-is and fine-tuning a new model; it offers a middle ground where the framework itself learns how to prompt the model better for your specific modules. Another strength is **reproducibility**: by treating LLM calls as functions with given signatures (and maybe seeding optimizations), one can get more consistent behavior across runs and less drifting of outputtheflyingbirds.intheflyingbirds.in】. The declarative approach also leads to clearer **design patterns** – instead of writing a lot of prompt strings, a developer writes Python code, which many find more maintainable. This is appealing for software engineering best practices (version control on code vs on prompt text, testing functions, etc.).

- *Weaknesses:* As of now, DSPy is more of a niche project and not as battle-tested as LangChain or SK. The user community is small, so resources and help may be limited. The framework's novel approach also has a learning curve: one has to learn the specific modules and the way DSPy wants you to structure a solution, which might be different from the more free-form prompting people are used to. Also, because it can do optimizations, using those features requires understanding some ML tuning concepts (though it automates a lot, one should know what objective to set, provide validation data, etc.). In dynamic agent scenarios, it's not clear how DSPy handles multi-step planning – it does have a ReAct module and multi-step support, but most examples are likely single-agent. Multi-agent or multi-tool interactions might be less emphasized than optimizing prompt programs for a single agent. Another consideration: *performance of optimization* – if you let DSPy tune prompts, that might require multiple trials (calls to LLM) which is a setup cost, though presumably done offline or infrequently.

Use Cases: DSPy is well-suited for scenarios where **accuracy and consistency are crucial**, and you're willing to invest effort upfront to optimize. For instance:

- **Fact-checking agent:** Suppose you have a pipeline that answers questions with citations. You could use DSPy to implement a `Predict` module (the LLM answer) and attach an evaluator that checks if the answer contains a citation for each statement, then let DSPy tune the prompt to maximize some correctness score. This can result in an agent that is far less likely to hallucinate unsupported facts (since it's penalized during tuning).

- **Content moderation or transformation tasks:** where output must meet certain criteria exactly (like converting text to JSON with specific schema, or extracting data). DSPy can optimize the prompt to minimize errors in extraction.

- **Enterprise QA with feedback loop:** If you have an internal evaluation (users rating answers or known ground-truth for some queries), you can feed that back into DSPy to self-improve the agent's performance over time – achieving a form of continuous learning at the prompt level.

- **Multi-stage reasoning tasks with verifiability:** e.g., an math problem solver agent that first plans (chain-of-thought) then computes an answer. You can separately optimize the planning part to ensure it's logically sound. In fact, DSPy was used to implement things like a complex browsing ReAct agent in a tutoriadspy.ailearnbybuilding.ai】, indicating it's capable of multi-step action sequences, not just single-turn prompts.

In a sense, DSPy is more akin to a **framework for advanced AI developers or researchers** who want fine-grained control and improvement of LLM behaviors. It might be overkill for simple use cases or early prototyping (where LangChain's quick tool integration shines), but for production systems that need to squeeze out reliability without training custom models from scratch, DSPy presents an intriguing solution.

Comparative Perspective

To sum up the comparisons:

- **LangChain** is like the Swiss army knife – lots of tools, easy to start, great for prototypes or broad use, but prompts can be somewhat implicit and trial-and-error.

- **CrewAI** is like a lean specialized toolkit – designed for orchestrating multiple agents efficiently, with more explicit roles and likely better performance control. Good for when you know you need multi-agent.

- **AutoGen** is the framework for multi-agent conversations – useful if your problem naturally breaks into a dialogue between AIs (or AI and human). It leverages the power of collaboration and is grounded in MSR research.

- **Semantic Kernel** is the enterprise workhorse – strongly structured, good for integration into existing apps, with an emphasis on maintainability, and soon to be enriched with research like AutoGen under the hood. Ideal for big organizations, especially those on Azure/.NET stack.

- **DSPy** is the optimizer and research platform – aimed at those who want to systematically improve their agent's performance and treat LLM programming more like a science/engineering discipline than an art. Suited for scenarios where correctness is paramount or for experimenting with novel prompting algorithms.

It's worth noting that these frameworks are not mutually exclusive. For example, one could use LangChain to prototype and then implement a refined version in Semantic Kernel for production. Or use DSPy's optimized prompts within a Semantic Kernel skill, etc. We also have other notable mentions: **LlamaIndex** (which focuses on document retrieval workflows but has agentic features in its "Agentic Query Engine"), **OpenAI's Function Calling/Agents SDK** (OpenAI has been releasing tools to make building simple agents easier, e.g., function calling could be seen as a mini-agent mechanism inside the model), and **Hugging Face Transformers Agent** (which allows tools usage with open models). The ecosystem is rapidly evolving.

For engineering managers and architects, the choice may come down to the stack compatibility and specific needs:

- If the team is Python-centric and wants quick results using GPT-4, LangChain or AutoGen might be first choices.

- If the project demands multi-agent or high throughput, CrewAI is attractive.

- If the company has a lot of C# expertise or needs deep Azure integration and longevity, Semantic Kernel is a strong candidate (especially as it matures its Agent Framework to GA with support from Microsoft).

- If the goal is to experiment with improving the model's reasoning quality or integrate with research efforts, DSPy is a great playground that could yield a more robust solution.

No single framework is "best" in all situations; they each carve out a niche. The good news is all are open-source (Semantic Kernel, LangChain, etc.) or at least accessible, so one can try a few and even mix ideas. As the field of agentic AI is new, these tools are actively adding features (for example, by 2025, LangChain also started offering a graph-based approach via LangGraplangfuse.com】 bridging some gap with Semantic Kernel's explicitness, and Semantic Kernel is integrating AutoGeinfoworld.com】). It's an exciting space to watch, and these frameworks themselves may converge or differentiate further in capabilities as time goes on.

7. Case Studies and Real-World Implementations

To ground the discussion, let's examine some real-world implementations of Agentic RAG. These case studies highlight how the concepts have been applied in practice, either in open-source projects, research prototypes, or deployed company solutions.

Case Study 1: Moveworks Enterprise AI Assistant

Moveworks, a company known for its AI chatbot for enterprise IT support, has described an agentic approach in its platform. The **Moveworks AI Assistant** is designed to answer employee questions and automate tasks by pulling information from a variety of enterprise systems (IT, HR, finance knowledge bases, ticketing systems, etc.moveworks.commoveworks.com】. Traditional chatbots were limited to a fixed knowledge base or a set of static rules, but Moveworks adopted an Agentic RAG architecture to handle more complex, dynamic queries.

How it works: When an employee asks a question, Moveworks' system will:

1. **Interpret the query** and break it into sub-tasks if needed. (For example, a query might need both a database lookup and a policy explanation.)

2. **Retrieve from multiple sources:** The agent can query the internal wiki, SharePoint documents, email knowledge base, or even live systems. Moveworks specifically notes their solution "combines LLM capabilities with information from curated knowledge sourcesmoveworks.com】. This implies an agent deciding which source is relevant. If the question is technical (e.g., about a software error), it might search in IT ticket logs; if it's HR-related, it searches policy docs.

3. **Use Tools/APIs:** If the query is an actionable request (like "reset my VPN password"), the agent can execute that action via integration (perhaps calling a backend API to trigger a password reset). Moveworks mentioned automating routine tasks as a benefit of their agentic approacmoveworks.com】.

4. **Iterative clarification:** If needed, the agent might ask the user a follow-up (though in many enterprise scenarios, the query is straightforward and context can be inferred from identity or previous interactions). The system maintains context about the user (role, department) as part of memory to tailor answers.

5. **Synthesize answer:** The LLM generates a response that may include information from multiple sources (with references or links). For example, *"According to the HR policy (see PolicyPortal link), you are entitled to X. Also, the last update on this from an all-hands email (see your inbox, Jan 5) mentioned Y."* This combines two pieces of info retrieved from different places.

Why agentic: Enterprise questions often cannot be answered by a single document. The agent's ability to fetch from various databases ensures higher answer accuracy and completenesmoveworks.com】. Moreover, if the question is multi-step (like "What's the status of my ticket and also how do I escalate it?"), a static system would fail. Moveworks' agent can get the status from the ticketing system (via API) and the escalation process from knowledge base, then compose the answer. This is essentially the agent performing two retrievals and merging the result, which is a mini-plan.

Moveworks also highlights **real-time adaptability**: if their knowledge sources update (say new policy rolled out), the agent draws on the latest data, making answers current (something users expect, as opposed to outdated static FAQs). They mention the assistant *"enhances overall productivity through efficient data management and retrieval”moveworks.com】 – in practice, their customers see employees getting answers in seconds to things that might have taken hours of searching or filing a help desk ticket.

From an architectural view, Moveworks’ implementation has a lot in common with our earlier generic description: an agent (the assistant) with tools (connectors to various enterprise data sources and action endpoints), memory (user context and past interactions), and an LLM orchestration that decides what to retrieve. While proprietary, the concepts line up with Agentic RAG: multi-step reasoning and tool use beyond a single knowledge base. It demonstrates that agentic systems are not just academic – they are deployed to thousands of users, answering diverse queries daily.

Case Study 2: Agentic Document Workflows (LlamaIndex)

】An example of an *Agentic Document Workflow* for document analysis. Here, an input document (e.g. an invoice or contract) is processed by an **Agent Workflow** that parses and extracts structured data, applies business rules and generates actions, all while maintaining state (context) across steps. The agent taps into a **Knowledge Base** (e.g., company databases or policy documents) for additional context, then produces **Structured Recommendations** such as summaries, compliance reports, or decisions. This real-world pattern shows an agent orchestrating multiple tools – document parsing, retrieval of relevant reference data, and rule-based analysis – to automate a complex document understanding task.

LlamaIndex (formerly GPT Index) is an open-source library focusing on connecting LLMs with external data. In late 2024, it introduced

Agentic Document Workflows (ADW)*llamaindex.ai】. This is essentially a case-specific Agentic RAG for processing documents in an end-to-end manner. Consider the use case of **contract review for compliance which we discussed conceptually; LlamaIndex actually showcased this pattern:

- **Input:** A large unstructured document (contract).

- ***Agent Orchestration:*** The ADW splits the task into sub-agents or sub-tasks. For instance, one part of the workflow uses a *parser* to extract structured info (clauses, entities like dates, parties), another part uses a *retriever* to get related info (like regulatory requirements or internal policy relevant to each clause), and then a *reasoning agent* assesses each clause against those requirementllamaindex.aillamaindex.ai】.

- **State Management:** The workflow maintains context – e.g., if a clause references something earlier, the agent remembers it as it checks later clauses. The system is aware of where it is in the document and what has been found so far.

- **Outcome:** A structured output such as a compliance report listing which clauses are acceptable and which are risky, along with suggested revisions.

One can view ADW as an *agent supervising a pipeline of tools*: an NLP parser tool, a vector search tool for knowledge base, and perhaps a rule-checking function. Rather than a human performing those steps manually (or writing separate scripts), the agent auto-coordinates them. LlamaIndex provides the template for this architecturllamaindex.aillamaindex.ai】, and it's been applied to things like automating financial document processing (invoices, insurance claims). For example, extracting line items from an invoice, checking them against a purchase order via a database lookup, then flagging discrepancies – an agent can handle that.

Anecdotally, companies have started using such agentic workflows in RPA (Robotic Process Automation) contexts. Instead of a rigid RPA script, an LLM agent can be more flexible in reading varied document formats and still perform the required actions. This is real-world value: reducing hours of human work reviewing documents down to seconds of AI work with a human just quickly verifying the AI's report.

Case Study 3: AutoGPT and Autonomous Web Research Agents

One of the phenomena of 2023 was the emergence of **AutoGPT** and similar autonomous research agents (like BabyAGI). These were community-driven projects that connected GPT-4 with a loop of self-prompting and tool use (especially web browsing) to attempt open-ended goals like "research this topic and formulate a business plan." While not always reliable, they were a compelling proof-of-concept of Agentic RAG capabilities and garnered widespread attention.

AutoGPT essentially created an agent that could:

- Create sub-goals from a high-level goal.

- Use a web search tool to gather information (retrieval).
- Use a text browser to read the content (further retrieval).
- Store notes (memory) and refine its plan.
- Continue this loop, generating “thoughts” and “actions” similar to a ReAct pattern, until it decided it had achieved the goal.

For example, if tasked with “find the best laptop under $1000 for programming and write a report,” AutoGPT would:

1. Search the web for “best programming laptops under $1000”.
2. Find some articles, read them.
3. Possibly search specific models or look for reviews.
4. Synthesize a report with pros/cons.

This is Agentic RAG with the web as the knowledge source. In practice, AutoGPT often got stuck or was inefficient (some jokingly called it “AutoGPT, the world’s fastest idiot, because it will tirelessly pursue something even if going in circles”). It sometimes looped or gave unconvincing results because it lacked human judgment. However, it demonstrated the *feasibility* of chaining multiple searches and operations to handle a complex research query without human intervention. It’s essentially a single-agent RAG router with a lot of freedom (maybe too much freedom, which was part of the issue).

The reception of AutoGPT in the real world showed both the promise and the current limitations:

- **Promise:** People saw that you could ask an AI agent to do a non-trivial project (like analyze a market and generate an outline of a strategy) and it would attempt it by gathering data from various sites, rather than just spitting out whatever was in its training data. This is powerful – it’s like having a junior analyst.

- **Limitations:** In many cases the results were off-mark or needed heavy validation. This highlighted issues like hallucination propagation (the agent might read a forum post that itself had incorrect info and then confidently report that as fact) and the challenge of open-ended goals (without a specific end criterion, the agent sometimes didn’t know when to stop).

Despite that, AutoGPT inspired a wave of similar projects and enhancements (some integrated vector databases to give the agent long-term memory of what it’s seen, to avoid revisiting the same info repeatedly). It also spurred work on **goal-oriented agent frameworks** (for example, integrating AutoGen’s ideas to have multiple agents instead of one self-loop).

From a research perspective, AutoGPT is a case study in the wild of an Agentic RAG trying to do autonomous Internet research. Companies took note – for instance, by early 2025, we see products like **Jasper’s**

web agent or others offering "internet browsing" modes for GPT-based assistants. Even OpenAI's own ChatGPT introduced a web browsing plugin (and more recently their GPTs with browsing built-in). These are essentially controlled versions of what AutoGPT attempted: using retrieval (web) to augment generation for open queries.

Lessons Learned: AutoGPT's saga underscores the importance of some guardrails: for instance, having a mechanism to stop an agent that's not making progress, and the value of multi-agent (maybe if it had a separate agent to critique its plan, it wouldn't loop as much). It also demonstrated that *the components largely work*: GPT-4 with a reasonably written prompt can use tools and carry context – a validation of the Agentic RAG concept. The shortcomings were more about strategy (knowing what to do) than the mechanism. In reaction, newer systems introduced explicit planning steps or hierarchical agents (just as our survey discussed in Hierarchical Agentic RAarxiv.org】).

Case Study 4: IBM watsonx.ai with Agentic RAG

IBM has been incorporating advanced RAG techniques into its **watsonx.ai** platform for enterprise AI. In an IBM blog/think piece, they gave an example of using their 8B parameter Granite model in an agentic RAG setup to answer complex queries by integrating external datarxiv.org】. While details are high-level, let's interpret what that means.

Imagine a client asks WatsonX: "Analyze the trend of supply chain disruptions in APAC in the last 5 years and how our company can mitigate related risks." This is complex – it needs data (maybe reports on supply chain disruptions) and analysis. A traditional approach might not handle it well. IBM's agentic system would do something like:

- Recognize it needs up-to-date info on supply chain disruptions -> use a retrieval plugin to get relevant excerpts from news

Case Study 4: IBM watsonx.ai – Multi-Source Enterprise QA

IBM's **watsonx.ai** platform has explored Agentic RAG to enhance its enterprise Q&A capabilities. IBM has reported that by leveraging an agentic approach with their Granite series LLM (e.g. Granite-3-8B-Instruct) and external data retrieval, they can handle complex user queries in a corporate setting that would otherwise stump a standalone modarxiv.org5】.

Consider an example (elaborative of IBM's notes): A user asks, "*What were the main supply chain disruptions in APAC over the last 5 years, and how might they affect our electronics business?*" Answering this thoroughly requires:

- Current and historical data on supply chain disruptions (perhaps from news or industry reports).

- Internal knowledge of the company's electronics business and its supply chain nodes.

- An analysis or synthesis connecting the two (how those disruptions could impact the company).

A watsonx.ai agentic solution might involve multiple specialized agents:

1. **External Data Agent:** Uses a web/news search tool or a curated database of news to retrieve information on APAC supply chain disruptions (factory fires, port closures, pandemics, etc. in the last 5 years). It might pull key statistics or events (e.g., "2021 semiconductor shortage due to X").

2. **Internal Data Agent:** Queries the company's internal knowledge bases – for example, reports on supply chain risk that the company created, or data about where the company's suppliers are located (to identify which of the disruptions from external data are relevant). This could involve searching an internal wiki or data warehouse.

3. **Coordinator/Analysis Agent:** Takes the findings from the external and internal agents and uses the LLM to synthesize an answer. It might say: *"Over the past 5 years, APAC has seen disruptions such as A, B,arxiv.org5】. Our company's supply chain is impacted at manufacturing sites in country X and port Y which correspond to those disruptions. Therefore, potential effects include ... and strategies to mitigate include ..."*. The agent ensures that specific data points (years, locations, impact percentages if any) are included – likely by having the external agent provide those as citations or notes.

What makes this agentic? The system is doing **multi-step, multi-source retrieval** – exactly what IBM highlights as a benefit:

allowing the LLM to pull information from multiple knowledge bases and tools to produce a context-aware answibm.com5】. IBM also emphasizes **accuracy** – an agentic system can validate or cross-check its results, whereas a traditional system might give an answer without knowing if it covered everythiibm.com8】. In our example, the agent could be instructed to ensure it addresses each part of the question (trends *and* how they affect the business). If an intermediate agent's output is lacking (say the internal data agent finds nothing about a certain disruption), the coordinator might prompt a re-search or at least note the uncertainty.

IBM's case study essentially demonstrates an **enterprise knowledge management assistant** (similar in spirit to the Moveworks case) but likely with even more emphasis on the agent making decisions about data sources and performing validation. They mention providing accurate, domain-specific answers by integrating external inarxiv.org5】 – implying the agent doesn't just answer from the LLM's training data but actively uses tools to get the latest or most relevant data, then the LLM (Granite model) composes the answer. Even though IBM's model is smaller (3-8B) compared to GPT-4, by grounding it via Agentic RAG, it can punch above its weight in terms of answer quality on specific queries.

This case also underscores the trend of companies combining proprietary models with agent frameworks: rather than rely on a single giant model, they orchestrate a set of components to achieve performance. It's likely IBM's implementation also logs the agent's decision process and interactions for audit – important in enterprise (e.g., knowing which documents were referenced to produce a given answer for compliance reasons).

Outcome: In field trials, such agent-enhanced systems have shown improved answer quality and user satisfaction for complex questions. They are more likely to say, for instance, "Based on data from Q3 2023, disruption X caused a 20% delay in deliveriarxiv.org5】, which could translate to a Y% inventory shortfall for our business unit" – concrete and actionable information – whereas a vanilla model might respond with generic statements. Essentially, IBM's use of Agentic RAG turns their AI from a generic chatbot into a specialized analyst that knows when and how to pull in evidence.

These case studies underscore that Agentic RAG is not just theoretical: it's being used in production (Moveworks, IBM) and was experimented with widely in the developer community (AutoGPT) as well as advanced open-source projects (LlamaIndex ADW). The common theme is that **agentic systems tackle tasks involving multiple steps or data sources that static systems handled poorly**. They bring us closer to autonomous assistants that can do a lot of the heavy lifting that knowledge workers or support staff traditionally did.

8. Future Directions and Research Opportunities

Agentic RAG is a fast-evolving field at the intersection of NLP, knowledge retrieval, and software engineering. While significant progress has been made in the last couple of years, there are many open research questions and opportunities for innovation. Here are some key future directions:

1. **Improved Planning and Strategy:** Today's agent planning (like ReAct prompting) is still fairly rudimentary. Agents often don't truly "plan" in a global sense; they make step-by-step decisions myopically. Future research may yield more sophisticated planning algorithms for agents – for example, using symbolic planning techniques or learned planners to map out a sequence of tool uses before executing. We might see hybrids where an LLM proposes a high-level plan (in natural language or pseudo-code) that is then verified or optimized by a secondary process before execution. This could make agents more efficient (fewer redundant steps) and less prone to getting stuck. There is already work on "reflective" agents that pause and reconsider if their approach is working (e.g., the Self-Refine or self-correct paradigmarxiv.orgarxiv.org7】 – this will likely expand. An interesting direction is having agents learn from experience which strategies succeed; for instance, an agent could remember that last time a certain plan failed and avoid it in future similar tasks, moving towards a form of meta-learning.

2. **Tool Learning and Expansion:** Currently, developers must explicitly add tools and define their interfaces for agents. In the future, agents might **learn to create their own tools or APIs on the fly**. For instance, if an agent determines it needs to perform a specific calculation repeatedly, a sufficiently advanced agent could generate a snippet of code as a new tool (some frameworks already let an agent write Python code and then execute it as needed – essentially creating custom tools at runtime). Another aspect is automatic tool selection: as

the number of available tools grows, agents need to choose relevant ones. Research could produce methods for the agent to intelligently query a “tool repository” (perhaps by reading documentation of tools) and pick the right sequence – akin to how humans consult software libraries. Microsoft’s integration of function calling in semantic kernel with a planner is a step in this direction, but there’s more to do to make it seamless. Additionally, we might see **standard tool APIs** emerge (for example, a common spec for a web search tool, or a calculator), and agents that can dynamically incorporate any tool adhering to those standards.

3. Better Memory and Long-Term Learning: Memory in current agents is often ephemeral or simplistic (storing the conversation or a vector database of past info). Future agent architectures may include more advanced memory systems:

- **Episodic memory:** the agent can remember past interactions or outcomes and apply lessons. For example, if an agent interacted with a user last week about a project, it should remember that context this week (without needing the user to restate it) – some systems do this with vector search over past dialogues, but there’s room for improvement in how memories are stored and retrieved (perhaps using event segmentation or summarization that’s sensitive to the agent’s goals).

- **Semantic memory / knowledge base integration:** an agent could build up its own knowledge graph of facts it has verified. We might see an agent that, after searching and reading, *asserts new facts* into a knowledge store for future use. This starts to blur into the area of continual learning – effectively the agent is curating data for itself.

- **Forgetting and Memory Management:** as agents run longer (some could run indefinitely as background processes), they'll need strategies to forget irrelevant details and compress important ones so they don't become overwhelmed or run into context window limits. Research into memory selection (what to keep vs drop) is crucial here.

4. Benchmarking and Evaluation Frameworks: Today, evaluating an agentic system is tricky. Traditional metrics like answer accuracy or BLEU for generation don't capture multi-step reasoning quality or tool use efficacy. The survey notes the lack of specialized benchmarks for agentic capabilities (like multi-agent collaboration, adaptabilitarxiv.orgarxiv.org3】. We can expect the community to develop **standard benchmark tasks** – for example, a benchmark where an agent must use a calculator to solve math problems, or one where an agent must gather clues from multiple documents to answer a question (to test multi-hop retrieval). There may also be benchmarks for multi-agent dialogue (like two agents solving a puzzle together). Having these will drive research progress and allow quantitative comparison of approaches. Additionally, evaluation metrics beyond "final answer correct" could be introduced: e.g., measuring the efficiency of an agent (did it use minimal steps?), coherence of chain-of-thought, or ability to recover from errors mid-way.

5. Safety, Ethics, and Alignment: As agents become more autonomous, ensuring they behave as intended is paramount. Future directions include:

- **Fine-grained control:** allowing human operators or system policies to constrain what an agent can do. For instance, an enterprise might want an agent never to call an external web search if the query is classified as sensitive. Research into *controllable agents* – where constraints can be injected (perhaps via secondary models that monitor the agent's decisions) – is likely.

- **Ethical decision-making:** Agents that perform actions should incorporate ethical reasoning. For example, a customer support agent might need to decide when to escalate to a human for empathy reasons. Embedding moral or policy-based heuristics into the planning process is a challenge (one approach could be to have a "guardian" agent that critiques the primary agent's decisions for compliance with ethical guidelinarxiv.orgarxiv.org1】).

- **Robustness to adversarial input:** Agents that use tools like web search could encounter adversarial content (imagine an agent reading a maliciously crafted webpage designed to trick LLMs). Research into hardening agents against such scenarios (maybe cross-verifying facts from multiple sources to avoid one poison pill source) will be important for security.

- **Transparency:** Another aspect is making agent decisions explainable. Future frameworks might automatically produce a user-friendly rationale of what the agent did ("I searched X because of Y, then I found Z which suggested..."). This builds trust and helps in debugging missteps. Some research is looking at summarizing the chain-of-thought for end-users without exposing the raw (and sometimes confusing) LLM thought process.

6. Domain-Specific Agentic RAG: We may see tailored agent architectures for specific domains:

- In **healthcare**, an agent might coordinate between medical literature, patient data, and protocol databases to provide decision support. But this needs heavy validation (perhaps a secondary agent that checks against guidelines to prevent unsafe suggestions). Research could focus on an agent that understands the boundaries of medical advice and when to defer to human doctors – essentially ensuring a *do-no-harm principle* in agent planning.

- In **finance**, agentic systems might be integrated with trading or risk management – here, an agent would need to operate under strict rules (to avoid, say, unauthorized trades). Future agent designs might include formal verification for certain critical actions or use sandboxed simulation environments to test an agent's plan before executing it live.

- In **education**, we might have multi-agent tutors (one plays student, one plays teacher to refine a solution, etc.) – research can explore how that improves learning outcomes. Also, agents that adapt to a student’s learning style over long-term interactions (learning the student’s weaknesses, etc., which ties back to better long-term memory).

7. Scaling and Efficiency Techniques: There is also the practical engineering angle: how to make agentic systems faster and cheaper. Some future directions:

- **Model specialization:** Using smaller specialized models for certain tools (e.g., use a lightweight model for planning, and a larger model for generation only when needed). This could drastically cut cost. For instance, a 2-step approach: a cheap model decides if it even needs to use the expensive GPT-4 for a query, or if a simpler answer suffices.

- **Parallelizing agents:** If an agent knows it needs to do two searches, doing them concurrently rather than sequentially can save latency. Frameworks might evolve to support parallel tool usage more naturally. However, parallel results need merging – so there’s research in how an agent can handle asynchronous information gathering.

- **On-device agents:** Running agents locally (on edge devices) is challenging now due to model size, but model compression and distillation could allow a smaller but still capable agent to run at least partially offline. This raises research questions: can we distill the reasoning behavior of a GPT-4 agent into a 13B parameter model so that it can run on-prem (for privacy or latency)? Some initial work on model distillation for chain-of-thought exists, and applying it to agent behavior is a promising avenue.

8. Combining Symbolic and Neural Methods: An intriguing direction is blending formal logic/symbolic reasoning with agentic LLMs. For example, an agent might convert certain problems into a symbolic form, solve them with a traditional algorithm, and convert back. Or maintain a knowledge base of facts that is updated with reasoning outputs (a bit like an evolving knowledge graph the agent trusts). The agent could use the symbolic module as a tool. This synergy can help ensure consistency (symbols don’t hallucinate) while retaining the flexibility of LLMs. Early examples include agents that use SQL engines via tools for any heavy data lifting, or use a theorem prover tool for structured logical inference. We can foresee more tight coupling, where the LLM delegates purely symbolic tasks to guarantee correctness (like complex math) and focuses on the connective tissue of reasoning.

In conclusion, Agentic RAG is on a trajectory to become more **autonomous, reliable, and integrated**. Each piece – from planning, to tool use, to memory – has active research aimed at making agents more like competent collaborators than experimental novelties. As benchmarks and real-world deployments inform what works and what doesn’t, we’ll see new patterns emerge (perhaps standardized agent architectures for certain classes of problems, similar to how design patterns emerged in software engineering).

For engineering managers and architects, the horizon suggests:

- Tools will become easier to plug and play, so investing in **infrastructure that can accommodate new agent capabilities** is wise (for example, designing systems with modularity so you can swap out the planner module when a better one comes).

- **Governance frameworks** around agents will likely develop (similar to MLOps for models, we'll have Ops for agents – monitoring their steps, outcomes, and intervening when needed).

- The distinction between an AI agent and a traditional software service may blur as agents can self-modify or improve. Policies on testing and validating any self-improvement will be crucial (one might require an agent's new "skill" to be approved before it's used in production, akin to a pull request from the AI).

It's an exciting time where foundational research in AI (like reinforcement learning, meta-learning, etc.) may find new application via these agent frameworks, and conversely, practical use of agents will drive new research questions. Agentic RAG systems stand to become a cornerstone of AI deployment in the coming years – evolving from today's semi-autonomous assistants to tomorrow's adaptive, context-savvy, and trustworthy autonomous collaborators. By staying abreast of these developments, engineering leaders can position their teams to leverage the next generation of AI capabilities, turning what are currently cutting-edge experiments into dependable components of enterprise software solutions.

References: (The references cited throughout the text correspond to sources that provide additional detail on the points discussed, including academic survearxiv.orgarxiv.org0】, industry bloibm.comglean.com1】, and documentation from frameworks like LangChaarxiv.org7】, Semantic Kerninfoworld.com7】, and others.)

PPT - Agentic RAG - A New Paradigm in Retrieval-Augmented Generation

Executive Summary

Agentic RAG introduces a significant evolution in Retrieval-Augmented Generation (RAG) by embedding autonomous agentic behavior into the RAG pipeline. This enables multi-step reasoning, self-initiated task planning, adaptive retrieval, and decision-making, resulting in more reliable, scalable, and context-aware outputs across various domains.

---

1. Introduction

Traditional RAG systems have improved factual grounding by retrieving relevant context from a knowledge base. However, they are often limited to single-turn interactions and static retrieval. Agentic RAG overcomes these limitations by integrating autonomous agents capable of planning, goal decomposition, iterative retrieval, and tool use.

---

2. What is Agentic RAG?

Agentic RAG is an evolution of RAG that incorporates agentic workflows, enabling:

- Multi-step planning and reasoning
- Dynamic tool usage (e.g., search, calculators, APIs)
- Iterative retrieval and self-reflection
- Task decomposition

- Memory and context persistence

These capabilities turn passive RAG into an autonomous, decision-making system.

---

3. Key Architectural Components

- **Planner**: Decomposes user intent into subgoals and tasks.
- **Retriever**: Fetches context dynamically at each step.
- **Executor**: Performs task-specific actions (e.g., calling APIs).
- **Memory Module**: Maintains context and past actions.
- **Critic/Reflector**: Evaluates output and decides whether to revise or proceed.

---

4. Comparison to Traditional RAG

Feature	Traditional RAG	Agentic RAG
Retrieval Frequency	One-time	Iterative
Task Complexity	Simple QA	Multi-step workflows
Autonomy	Low	High

Tool Use	Rare	Integrated
Reflection/Correction	None	Built-in

---

5. Why Agentic RAG Matters

- **Reliability**: Reduces hallucination by verifying with tools and memory.
- **Scalability**: Modular tasks handled by different agents.
- **Complex Tasks**: Can handle reasoning-heavy queries (e.g., legal, financial, code).
- **Contextual Awareness**: Persistent memory allows deeper understanding over time.

---

6. Use Cases

- **Enterprise Q&A**: Handle complex, multi-document queries with citations.
- **DevOps Copilot**: Troubleshoot and take actions across logs, metrics, APIs.
- **Legal Assistant**: Extract clauses, validate with precedents, explain in layman terms.
- **Education**: Tutor that adapts to learning progress and student knowledge gaps.

---

7. Implementation Considerations

- **LLM Selection**: Must support function calling, multi-turn context.
- **Tooling**: Integration with search APIs, calculators, databases.
- ***Agent Frameworks***: LangGraph, CrewAI, AutoGen, etc.
- **Memory Stores**: Vector DBs (Weaviate, Qdrant), relational for tool outputs.
- **Evaluation**: Needs new benchmarks for agent workflows.

---

8. Challenges

- Cost and latency from multi-step operations
- Debugging and traceability of agent decisions
- Benchmarking is nascent
- Orchestration across multiple tools and agents

---

9. Future Directions

- Multimodal Agentic RAG (image, audio input)
- Hierarchical agents with specializations
- Long-term memory and personalization
- Agent-RAG hybrid with symbolic reasoning

---

10. Conclusion

Agentic RAG represents the next frontier in RAG-based systems, offering autonomous, reliable, and extensible generation capabilities. It opens new possibilities for high-stakes, complex tasks that require not just facts but also reasoning, planning, and adaptability.