Level Up Coding · Follow publication

Member-only story

# Building an Agentic Deep-Thinking RAG Pipeline to Solve Complex Queries

Planning, Retrieval, Reflection, Critique, Synthesis and more

68 min read · 4 days ago

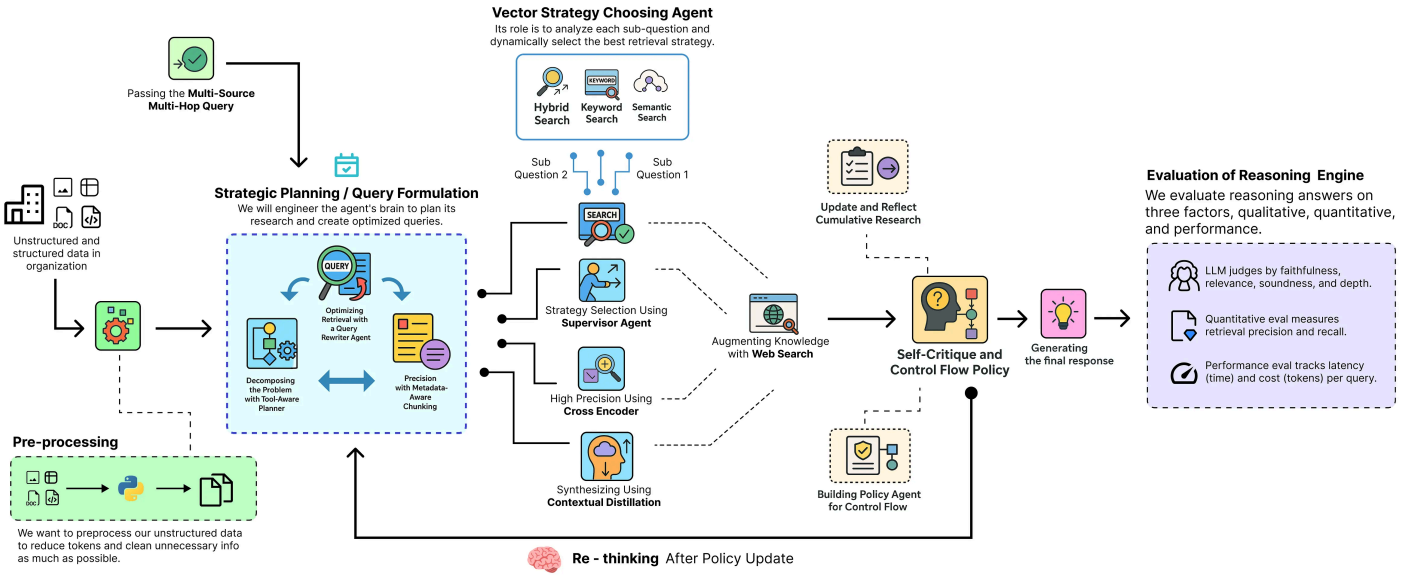
Fareed Khan Follow



Read this story for free: link

A RAG system often fails not because the LLM lacks intelligence, but because its **architecture is too simple**. It tries to handle a cyclical, multi-step problem with a linear, one-shot approach.

Many complex queries demand **reasoning**, **reflection**, and **smart decisions** about when to act, much like how we retrieve information when faced with a question. That's where agent-driven actions within the RAG pipeline come into play. Let's take a look at what a typical deep-thinking RAG pipeline looks like...




Deep Thinking RAG Pipeline (Created by Fareed Khan)

1. **Plan:** First, the agent decomposes the complex user query into a structured, multi-step research plan, deciding which tool (internal document search or web search) is needed for each step.

2. **Retrieve:** For each step, it executes an adaptive, multi-stage retrieval funnel, using a supervisor to dynamically choose the best search strategy (vector, keyword, or hybrid).

3. **Refine:** It then uses a high-precision cross-encoder to rerank the initial results and a distiller agent to compress the best evidence into a concise context.

4. **Reflect:** After each step, the agent summarizes its findings and updates its research history, building a cumulative understanding of the problem.

5. **Critique:** A policy agent then inspects this history, making a strategic decision to either continue to the next research step, revise its plan if it hits a dead end, or finish.

6. **Synthesize:** Once the research is complete, a final agent synthesizes all the gathered evidence from all sources into a single, comprehensive, and citable answer.

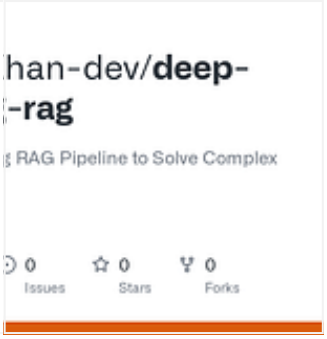
In this blog, we are going to implement the entire **deep thinking RAG pipeline** and compare it with a basic RAG pipeline to demonstrate how it solves **complex multi-hop queries**.

All the code + theory is available in my GitHub Repository:

**GitHub - FareedKhan-dev/deep-thinking-rag: A Deep Thinking RAG Pipeline to Solve Complex Queries**

A Deep Thinking RAG Pipeline to Solve Complex Queries - GitHub - FareedKhan-dev/deep-thinking-rag: A Deep Thinking RAG...

github.com




## Table of Contents


- Setting up the Environment
- Sourcing the Knowledge Base
- Understanding our Multi-Source, Multi-Hop Query
- Building a Shallow RAG Pipeline that will Fail
- Defining the RAG State for Central Agent System
- Strategic Planning and Query Formulation
  - Decomposing the Problem with Tool-Aware Planner
  - Optimizing Retrieval with a Query Rewriter Agent
  - Precision with Metadata-Aware Chunking
- Creating The Multi-Stage Retrieval Funnel
  - Dynamically Choosing a Strategy Using Supervisor
  - Broad Recall with Hybrid, Keyword and Semantic Search
  - High Precision Using a Cross-Encoder Reranker
  - Synthesizing using Contextual Distillation
- Augmenting Knowledge with Web Search
- Self-Critique and Control Flow Policy
  - Update and Reflect Cumulative Research History
  - Building Policy Agent for Control Flow
- Defining the Graph Nodes
- Defining the Conditional Edges
- Wiring the Deep Thinking RAG Machine
- Compiling and Visualizing the Iterative Workflow

- Running the Deep Thinking Pipeline
- Analyzing the Final, High-Quality Answer
- Side by Side Comparison
- Evaluation Framework and Analyzing Results
- Summarizing Our Entire Pipeline
- Learned Policies with Markov Decision Processes (MDP)


. . .

### Setting up the Environment

Before we can start coding the Deep RAG pipeline, we need to begin with a strong foundation because a production-grade AI system is not only about the final algorithm, it is also about the deliberate choices we make during setup.

> Each of the steps we are going to implement is important in determining how effective and reliable the final system will be.

When we start developing a pipeline and performing trial and error with it, it's better to define our configuration in a plain dictionary format because later on, when the pipeline gets complicated, we can simply refer back to this dictionary to change the config and see its impact on the overall performance.

```
# Central Configuration Dictionary to manage all system parameters
config = {
    "data_dir": "./data",                           # Directory to store raw an
    "vector_store_dir": "./vector_store",           # Directory to persist our
    "llm_provider": "openai",                       # The LLM provider we are u
    "reasoning_llm": "gpt-4o",                      # The powerful model for pl
    "fast_llm": "gpt-4o-mini",                      # A faster, cheaper model f
    "embedding_model": "text-embedding-3-small",    # The model for creating do
    "reranker_model": "cross-encoder/ms-marco-MiniLM-L-6-v2", # The model for p
    "max_reasoning_iterations": 7,                  # A safeguard to prevent th
    "top_k_retrieval": 10,                          # Number of documents for i
```

```
    "top_n_rerank": 3,                                  # Number of documents to ke
}
```

These keys are pretty easy to understand but there are three keys that are worth mentioning:

- `llm_provider`: This is the LLM provider we are using, in this case, OpenAI. I am using OpenAI because we can easily swap models and providers in LangChain, but you can choose any provider that suits your needs like Ollama.
- `reasoning_llm`: This must be the most powerful in our entire setup because it will be used for planning and synthesis.
- `fast_llm`: This should be a faster and cheaper model because it will be used for simpler tasks like the baseline RAG.

Now we need to import the required libraries that we will be using through our pipeline along with setting the api keys as an environment variabls to avoid exposing it in the code blocks.

```
import os                   # For interacting with the operating system (e.g., m
import re                   # For regular expression operations, useful for text
import json                 # For working with JSON data
from getpass import getpass # To securely prompt for user input like API keys w
from pprint import pprint   # For pretty-printing Python objects, making them m
import uuid                 # To generate unique identifiers
from typing import List, Dict, TypedDict, Literal, Optional # For type hinting

# Helper function to securely set environment variables if they are not already
def _set_env(var: str):
    # Check if the environment variable is not already set
    if not os.environ.get(var):
        # If not, prompt the user to enter it securely
        os.environ[var] = getpass(f"Enter your {var}: ")

# Set the API keys for the services we will use
_set_env("OPENAI_API_KEY")      # For accessing OpenAI models (GPT-4o, embeddin
_set_env("LANGSMITH_API_KEY")   # For tracing and debugging with LangSmith
_set_env("TAVILY_API_KEY")      # For the web search tool

# Enable LangSmith tracing to get detailed logs and visualizations of our agent
os.environ["LANGSMITH_TRACING"] = "true"
```

```
# Define a project name in LangSmith to organize our runs
os.environ["LANGSMITH_PROJECT"] = "Advanced-Deep-Thinking-RAG"
```

We are also enabling LangSmith for tracing. When you are working with an agentic system that has a complex, cyclical workflow, **tracing is not just a nice-to-have it's important**. It helps you visualize what's going on and makes it much easier to debug the agent's thought process.

### Sourcing the Knowledge Base

A production-grade RAG system requires a knowledge base that is both complex and demanding in order to truly demonstrate its effectiveness. For this purpose, we will use **NVIDIA's 2023 10-K filing**, a comprehensive document exceeding one hundred pages that details the company business operations, financial performance, and disclosed risk factors.

Sourcing the Knowledge Base (Created by Fareed Khan)

First, we will implement a **custom function** that programmatically downloads the 10-K filing directly from the **SEC EDGAR database**, parses the raw HTML, and converts it into a clean and structured text format suitable for ingestion by our RAG pipeline. So let's code that function.

```
import requests # For making HTTP requests to download the document
from bs4 import BeautifulSoup # A powerful library for parsing HTML and XML doc
from langchain.docstore.document import Document # LangChain's standard data st
```

```
def download_and_parse_10k(url, doc_path_raw, doc_path_clean):
    # Check if the cleaned file already exists to avoid re-downloading
    if os.path.exists(doc_path_clean):
        print(f"Cleaned 10-K file already exists at: {doc_path_clean}")
        return

    print(f"Downloading 10-K filing from {url}...")
    # Set a User-Agent header to mimic a browser, as some servers block scripts
    headers = {'User-Agent': 'Mozilla/5.0'}
    # Make the GET request to the URL
    response = requests.get(url, headers=headers)
    # Raise an error if the download fails (e.g., 404 Not Found)
    response.raise_for_status()

    # Save the raw HTML content to a file for inspection
    with open(doc_path_raw, 'w', encoding='utf-8') as f:
        f.write(response.text)
    print(f"Raw document saved to {doc_path_raw}")

    # Use BeautifulSoup to parse and clean the HTML content
    soup = BeautifulSoup(response.content, 'html.parser')

    # Extract text from common HTML tags, attempting to preserve paragraph stru
    text = ''
    for p in soup.find_all(['p', 'div', 'span']):
        # Get the text from each tag, stripping extra whitespace, and add newli
        text += p.get_text(strip=True) + '\n\n'

    # Use regex to clean up excessive newlines and spaces for a cleaner final t
    clean_text = re.sub(r'\n{3,}', '\n\n', text).strip() # Collapse 3+ newlines
    clean_text = re.sub(r'\s{2,}', ' ', clean_text).strip() # Collapse 2+ space

    # Save the final cleaned text to a .txt file
    with open(doc_path_clean, 'w', encoding='utf-8') as f:
        f.write(clean_text)
    print(f"Cleaned text content extracted and saved to {doc_path_clean}")
```

The code is pretty easy to understand, we are using beautifulsoup4 to parse the HTML content and extract the text. It will help us to easily navigate the HTML structure and retrieve the relevant information while ignoring any unnecessary elements like scripts or styles.

Now, let's execute this and see how it works.

```
print("Downloading and parsing NVIDIA's 2023 10-K filing...")
# Execute the download and parsing function
```

```
download_and_parse_10k(url_10k, doc_path_raw, doc_path_clean)

# Open the cleaned file and print a sample to verify the result
with open(doc_path_clean, 'r', encoding='utf-8') as f:
    print("\n--- Sample content from cleaned 10-K ---")
    print(f.read(1000) + "...")


#### OUTPUT ####
Downloading and parsing NVIDIA 2023 10-K filing...
Successfully downloaded 10-K filing from https://www.sec.gov/Archives/edgar/dat
Raw document saved to ./data/nvda_10k_2023_raw.html
Cleaned text content extracted and saved to ./data/nvda_10k_2023_clean.txt

# --- Sample content from cleaned 10-K ---
Item 1. Business.
 OVERVIEW
 NVIDIA is the pioneer of accelerated computing. We are a full-stack computing
 Founded in 1993, we started as a PC graphics chip company, inventing the graph
 The programmability of our GPUs made them ...
```

We are simply calling this function storing all the content in a txt file that will serve as our context for our rag pipeline.

When we run the above code you can see that it starts download the report for us and we can see how a sample of our downloaded content looks like.

### Understanding our Multi-Source, Multi-Hop Query

To test our implemented pipeline and compare it with basic RAG, we need to use a very complex query that covers different aspects of the documents we are working with.

```
Our Complex Query:

"Based on NVIDIA's 2023 10-K filing, identify their key risks related to
competition. Then, find recent news (post-filing, from 2024) about AMD's
AI chip strategy and explain how this new strategy directly addresses or
exacerbates one of NVIDIA's stated risks."
```

Let's break down why this query is so difficult for a standard RAG pipeline:

1. **Multi-Hop Reasoning:** It cannot be answered in a single step. The system must first identify the risks, then find the AMD news, and finally synthesize the two.
2. **Multi-Source Knowledge:** The required information lives in two completely different places. The risks are in our static, internal document (the 10-K), while the AMD news is external and requires access to the live web.
3. **Synthesis and Analysis:** The query doesn't ask for a simple list of facts. It demands an explanation of how one set of facts makes worse another, a task that requires true synthesis.

In the next section we are going to implement basic RAG pipeline actually see how simple RAG is failing this.

### Building a Shallow RAG Pipeline that will Fail

Now that we have our environment configured and our challenging knowledge base ready, our next logical step is to build a standard **vanilla** RAG pipeline. This serves a critical purpose …

> First building the simplest possible solution, we can run our complex query against it and observe exactly how and why it fails.

Here's what we are going to do in this section:

Shallow RAG Pipeline (Created by Fareed Khan)

- **Load and Chunk the Document:** We will ingest our cleaned 10-K filing and split it into small, fixed-size chunks a common but semantically naive approach.
- **Create a Vector Store:** We then are going to embed these chunks and index them in a ChromaDB vector store to enable basic semantic search.
- **Assemble the RAG Chain:** We will be using LangChain Expression Language (LCEL), which will wire together our retriever, a prompt template, and an LLM into a linear pipeline.
- **Demonstrate the Critical Failure:** We will execute our multi-hop, multi-source query against this simple system and analyze its inadequate response.

First, we need to load our cleaned document and split it. We will use the `RecursiveCharacterTextSplitter`, a standard tool in the LangChain ecosystem.

```
from langchain_community.document_loaders import TextLoader # A simple loader f
from langchain.text_splitter import RecursiveCharacterTextSplitter # A standard

print("Loading and chunking the document...")
# Initialize the loader with the path to our cleaned 10-K file
loader = TextLoader(doc_path_clean, encoding='utf-8')
# Load the document into memory
documents = loader.load()

# Initialize the text splitter with a defined chunk size and overlap
# chunk_size=1000: Each chunk will be approximately 1000 characters long.
# chunk_overlap=150: Each chunk will share 150 characters with the previous one
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=1
# Split the loaded document into smaller, manageable chunks
doc_chunks = text_splitter.split_documents(documents)

print(f"Document loaded and split into {len(doc_chunks)} chunks.")


#### OUTPUT ####
Loading and chunking the document...
Document loaded and split into 378 chunks.
```

We are having `378` chunks on our main doc, the next step is to make them searchable. For this, we need to create vector embeddings and store them in a

database. We will use ChromaDB, a popular in-memory vector store, and OpenAI `text-embedding-3-small` model as defined in our config.

```
from langchain_community.vectorstores import Chroma # The vector store we will
from langchain_openai import OpenAIEmbeddings # The function to create embeddin

print("Creating baseline vector store...")
# Initialize the embedding function using the model specified in our config
embedding_function = OpenAIEmbeddings(model=config['embedding_model'])

# Create the Chroma vector store from our document chunks
# This process takes each chunk, creates an embedding for it, and indexes it.
baseline_vector_store = Chroma.from_documents(
    documents=doc_chunks,
    embedding=embedding_function
)
# Create a retriever from the vector store
# The retriever is the component that will actually perform the search.
# search_kwargs={"k": 3}: This tells the retriever to return the top 3 most rel
baseline_retriever = baseline_vector_store.as_retriever(search_kwargs={"k": 3})

print(f"Vector store created with {baseline_vector_store._collection.count()} e


#### OUTPUT ####
Creating baseline vector store...
Vector store created with 378 embeddings.
```

`Chroma.from_documents` organize this process and stores all the vectors in an searchable index. The final step is to assemble them into a single, runnable RAG chain using LangChain Expression Language (LCEL).

This chain will define the linear flow of data: from the user' question to the retriever, then to the prompt, and finally to the LLM.

```
from langchain_core.prompts import ChatPromptTemplate # For creating prompt tem
from langchain_openai import ChatOpenAI # The OpenAI chat model interface
from langchain_core.runnable import RunnablePassthrough # A tool to pass inputs
from langchain_core.output_parsers import StrOutputParser # To parse the LLM's

# This template instructs the LLM on how to behave.
# {context}: This is where we will inject the content from our retrieved docume
# {question}: This is where the user's original question will go.
```

```
template = """You are an AI financial analyst. Answer the question based only o
{context}

Question: {question}
"""
prompt = ChatPromptTemplate.from_template(template)
# We use our 'fast_llm' for this simple task, as defined in our config
llm = ChatOpenAI(model=config["fast_llm"], temperature=0)

# A helper function to format the list of retrieved documents into a single str
def format_docs(docs):
    return "\n\n---\n\n".join(doc.page_content for doc in docs)

# The complete RAG chain defined using LCEL's pipe (|) syntax
baseline_rag_chain = (
    # The first step is a dictionary that defines the inputs to our prompt
    {"context": baseline_retriever | format_docs, "question": RunnablePassthrou
    # The context is generated by taking the question, passing it to the retrie
    # The original question is passed through unchanged
    | prompt # The dictionary is then passed to the prompt template
    | llm # The formatted prompt is passed to the language model
    | StrOutputParser() # The LLM's output message is parsed into a string
)
```

You do know that we define a dictionary as the first step. Its `context` key is populated by a sub-chain, the input question goes to the `baseline_retriever`, and its output (a list of `Document` objects) is formatted into a single string by `format_docs`. The `question` key is populated by simply passing the original input through using `RunnablePassthrough`.

Let's run this simple pipeline and understand where it is failing.

```
from rich.console import Console # For pretty-printing output with markdown
from rich.markdown import Markdown

# Initialize the rich console for better output formatting
console = Console()

# Our complex, multi-hop, multi-source query
complex_query_adv = "Based on NVIDIA's 2023 10-K filing, identify their key ris

print("Executing complex query on the baseline RAG chain...")
# Invoke the chain with our challenging query
baseline_result = baseline_rag_chain.invoke(complex_query_adv)
```

```
console.print("\n--- BASELINE RAG FAILED OUTPUT ---")
# Print the result using markdown formatting for readability
console.print(Markdown(baseline_result))
```

When you run the above code we get the following output.

```
#### OUTPUT ####
Executing complex query on the baseline RAG chain...

--- BASELINE RAG FAILED OUTPUT ---
Based on the provided context, NVIDIA operates in an intensely competitive semi
industry and faces competition from companies like AMD. The context mentions
that the industry is characterized by rapid technological change. However, the
```

There are three things that you might have notice in this failed RAG pipeline and its output.

- **Irrelevant Context:** Retriever grabs general chunks on **“NVIDIA”**, **“competition”** and **“AMD”** but misses specific 2024 AMD strategy details.
- **Missing Information:** Key failure is that 2023 data can’t cover 2024 events. System doesn’t realize it’s lacking crucial info.
- **No Planning or Tool Use:** Treats complex query as simple. Can’t break it into steps or use tools like web search to fill gaps.

> The system failed not because the LLM was dumb but because the architecture was too simple. It was a linear, one-shot process trying to solve a cyclical, multi-step problem.

Now that we have understand the issues with our basic RAG pipeline we can now start implementing our deep thinking methodology and see how well it solves our complex query.

### Defining the `RAG State for` Central Agent System

To build our reasoning agent, we first need a way to manage its state. In our simple RAG chain, each step was stateless, but …

> an intelligent agent, however, needs a memory. It needs to remember the original question, the plan it created, and the evidence it has gathered so far.

RAG State (Created by Fareed Khan)

The `RAGState` will act as a central memory, passed between every node in our LangGraph workflow. To build it, we will define a series of structured data classes, starting with the most fundamental building block: a single step in a research plan.

We want to define the atomic unit of our agent's plan. Each `Step` must contain not just a question to be answered, but also the reasoning behind it and, crucially, the specific tool the agent should use. This forces the agent's planning process to be explicit and structured.

```
from langchain_core.documents import Document
from langchain_core.pydantic_v1 import BaseModel, Field

# Pydantic model for a single step in the agent's reasoning plan
class Step(BaseModel):
    # A specific, answerable sub-question for this research step
    sub_question: str = Field(description="A specific, answerable question for
    # The agent's justification for why this step is necessary
    justification: str = Field(description="A brief explanation of why this ste
    # The specific tool to use for this step: either internal document search o
```

```
    tool: Literal["search_10k", "search_web"] = Field(description="The tool to
    # A list of critical keywords to improve the accuracy of the search
    keywords: List[str] = Field(description="A list of critical keywords for se
    # (Optional) A likely document section to perform a more targeted, filtered
    document_section: Optional[str] = Field(description="A likely document sect
```

Our `Step` class, using Pydantic `BaseModel`, acts as a strict contract for our Planner Agent. The `tool: Literal[...]` field forces the LLM to make a concrete decision between using our internal knowledge (`search_10k`) or seeking external information (`search_web`).

> This structured output is far more reliable than trying to parse a natural language plan.

Now that we have defined a single `Step`, we need a container to hold the entire sequence of steps. We will create a `Plan` class that is simply a list of `Step` objects. This represents the agent complete, end-to-end research strategy.

```
# Pydantic model for the overall plan, which is a list of individual steps
class Plan(BaseModel):
    # A list of Step objects that outlines the full research plan
    steps: List[Step] = Field(description="A detailed, multi-step plan to answe
```

We coded a `Plan` class that is going to provide the structure for the entire research process. When we invoke our Planner Agent, we will ask it to return a JSON object that conforms to this schema. This make sure that the agent strategy is clear, sequential, and machine-readable before any retrieval actions are taken.

Next, as our agent executes its plan, it needs a way to remember what it has learned. We will define a `PastStep` dictionary to store the results of each completed step. This will form the agent's **research history** or **lab notebook**.

```
# A TypedDict to store the results of a completed step in our research history
class PastStep(TypedDict):
```

```
    step_index: int               # The index of the completed step (e.g., 1, 2,
    sub_question: str             # The sub-question that was addressed in this
    retrieved_docs: List[Document] # The precise documents retrieved and rerank
    summary: str                  # The agent's one-sentence summary of the find
```

This `PastStep` structure is crucial for the agent's self-critique loop. After each step, we will populate one of these dictionaries and add it to our state. The agent will then be able to review this growing list of summaries to understand what it knows and decide if it has enough information to finish its task.

Finally, we will bring all these pieces together into the master `RAGState` dictionary. This is the central object that will flow through our entire graph, holding the original query, the full plan, the history of past steps, and all the intermediate data for the *current* step being executed.

```
# The main state dictionary that will be passed between all nodes in our LangGr
class RAGState(TypedDict):
    original_question: str     # The initial, complex query from the user that
    plan: Plan                 # The multi-step plan generated by the Planner A
    past_steps: List[PastStep] # A cumulative history of completed research ste
    current_step_index: int    # The index of the current step in the plan bein
    retrieved_docs: List[Document] # Documents retrieved in the current step (r
    reranked_docs: List[Document]  # Documents after precision reranking in the
    synthesized_context: str   # The concise, distilled context generated from
    final_answer: str          # The final, synthesized answer to the user's or
```

This `RAGState` `TypedDict` is the complete **mind** of our agent. Every node in our graph will receive this dictionary as input and return an updated version of it as output.

For example, the `plan_node` will populate the `plan` field, the `retrieval_node` will populate the `retrieved_docs` field, and so on. This shared, persistent state is what enables the complex, iterative reasoning that our simple RAG chain lacked.

With the blueprint for our agent's memory now defined, we are ready to build the first cognitive component of our system: the Planner Agent that will populate this state.

### Strategic Planning and Query Formulation

With our `RAGState` defined, we can now build the first and arguably most critical cognitive component of our agent: its ability to **plan**. This is where our system makes the leap from a simple data fetcher to a true reasoning engine. Instead of naively treating the user's complex query as a single search, our agent will first pause, think, and construct a detailed, step-by-step research strategy.

Strategic Planning (Created by Fareed Khan)

This section is broken down into three key engineering steps:

- **The Tool-Aware Planner:** We will build an LLM-powered agent whose sole job is to decompose the user's query into a structured `Plan` object, deciding which tool to use for each step.
- **The Query Rewriter:** We'll create a specialized agent to transform the planner's simple sub-questions into highly effective, optimized search queries.
- **Metadata-Aware Chunking:** We will re-process our source document to add section-level metadata, a crucial step that unlocks high-precision, filtered retrieval.

#### Decomposing the Problem with Tool-Aware Planner

So, basically we want to build the **brain of our operation**. The first thing this brain needs to do when it gets a complex question is to figure out a game plan.

Decomposing Step (Created by Fareed Khan)

> We can't just throw the whole question at our database and hope for the best. We need to teach the agent how to break the problem down into smaller, manageable pieces.

To do this, we will create a dedicated **Planner Agent**. We need to give it a very clear set of instructions, or a prompt, that tells it exactly what its job is.

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from rich.pretty import pprint as rprint

# The system prompt that instructs the LLM how to behave as a planner
planner_prompt = ChatPromptTemplate.from_messages([
    ("system", """You are an expert research planner. Your task is to create a
You have two tools available:
1. `search_10k`: Use this to search for information within NVIDIA's 2023 10-K f
2. `search_web`: Use this to search the public internet for recent news, compet
Decompose the user's query into a series of simple, sequential sub-questions. F
For `search_10k` steps, also identify the most likely section of the 10-K (e.g.
It is critical to use the exact section titles found in a 10-K filing where pos
```

```
        ("human", "User Query: {question}") # The user's original, complex query
    ])
```

We are basically giving the LLM a new persona: an **expert research planner**. We explicitly tell it about the two tools it has at its disposal (`search_10k` and `search_web`) and give it guidance on when to use each one. This is the "tool-aware" part.

We are not just asking it for a plan but asking it to create a plan that maps directly to the capabilities we have built.

Now we can initiate the reasoning model and chain it together with our prompt. A very important step here is to tell the LLM that its final output must be in the format of our Pydantic `Plan` class. This makes the output structured and predictable.

```
# Initialize our powerful reasoning model, as defined in the config
reasoning_llm = ChatOpenAI(model=config["reasoning_llm"], temperature=0)

# Create the planner agent by piping the prompt to the LLM and instructing it t
planner_agent = planner_prompt | reasoning_llm.with_structured_output(Plan)
print("Tool-Aware Planner Agent created successfully.")

# Let's test the planner agent with our complex query to see its output
print("\n--- Testing Planner Agent ---")
test_plan = planner_agent.invoke({"question": complex_query_adv})

s# Use rich's pretty print for a clean, readable display of the Pydantic object
rprint(test_plan)
```

We take our `planner_prompt`, pipe it to our powerful `reasoning_llm`, and then use the `.with_structured_output(Plan)` method. This tells LangChain to use the model function-calling abilities to format its response as a JSON object that perfectly matches our `Plan` Pydantic schema. This is much more reliable than trying to parse a plain text response.

Let's look at the output when we test it with our challenge query.

```
#### OUTPUT ####
Tool-Aware Planner Agent created successfully.

--- Testing Planner Agent ---
Plan(
|   steps=[
|   |   Step(
|   |   |   sub_question="What are the key risks related to competition as stat
|   |   |   justification="This step is necessary to extract the foundational i
|   |   |   tool='search_10k',
|   |   |   keywords=['competition', 'risk factors', 'semiconductor industry',
|   |   |   document_section='Item 1A. Risk Factors'
|   |   ),
|   |   Step(
|   |   |   sub_question="What are the recent news and developments in AMD's AI
|   |   |   justification="This step requires finding up-to-date, external info
|   |   |   tool='search_web',
|   |   |   keywords=['AMD', 'AI chip strategy', '2024', 'MI300X', 'Instinct ac
|   |   |   document_section=None
|   |   )
|   ]
)
```

If we look at the output you can see that the agent didn't just give us a vague plan, it produced a structured `Plan` object. It correctly identified that the query has two parts.

1. For the first part, it knew the answer was in the 10-K and chose the `search_10k` tool, even correctly guessing the right document section.
2. For the second part, it knew "news from 2024" couldn't be in a 2023 document and correctly chose the `search_web` tool. This is the first sign our pipeline will give promising result at least in the thinking process.

#### Optimizing Retrieval with a Query Rewriter Agent

So, basically we have a plan with good sub-questions.

> But a question like "What are the risks?" isn't a great search query. It's too generic. Search engines, whether they are vector databases or web search, work best with specific, keyword-rich queries.

Query Rewriting Agent (Creted by Fareed Khan)

To fix this, we will build another small, specialized agent: the **Query Rewriter**. Its only job is to take the sub-question for the current step and make it better for searching by adding relevant keywords and context from what we've already learned.

First, let's design the prompt for this new agent.

```
from langchain_core.output_parsers import StrOutputParser # To parse the LLM's

# The prompt for our query rewriter, instructing it to act as a search expert
query_rewriter_prompt = ChatPromptTemplate.from_messages([
    ("system", """You are a search query optimization expert. Your task is to r
The rewritten query should be specific, use terminology likely to be found in t
    ("human", "Current sub-question: {sub_question}\n\nRelevant keywords from p
])
```

We are basically telling this agent to act like a **search query optimization expert**. We are giving it three pieces of information to work with: the simple `sub_question`, the `keywords` our planner already identified, and the `past_context` from any previous research steps. This gives it all the raw material it needs to construct a much better query.

Now we can initiate this agent. It's a simple chain since we just need a string as output.

```
# Create the agent by piping the prompt to our reasoning LLM and a string outpu
query_rewriter_agent = query_rewriter_prompt | reasoning_llm | StrOutputParser(
print("Query Rewriter Agent created successfully.")

# Let's test the rewriter agent. We'll pretend we've already completed the firs
print("\n--- Testing Query Rewriter Agent ---")

# Let's imagine we are at a final synthesis step that needs context from the fi
test_sub_q = "How does AMD's 2024 AI chip strategy potentially exacerbate the c
test_keywords = ['impact', 'threaten', 'competitive pressure', 'market share',

# We create some mock "past context" to simulate what the agent would know at t
test_past_context = "Step 1 Summary: NVIDIA's 10-K lists intense competition an

# Invoke the agent with our test data
rewritten_q = query_rewriter_agent.invoke({
    "sub_question": test_sub_q,
    "keywords": test_keywords,
    "past_context": test_past_context
})

print(f"Original sub-question: {test_sub_q}")
print(f"Rewritten Search Query: {rewritten_q}")
```

To test this properly, we have to simulate a real scenario. We create a `test_past_context` string that represents the summaries the agent would have already generated from the first two steps of its plan. Then we feed this, along with the next sub-question, to our `query_rewriter_agent`.

Let's look at the result.

```
#### OUTPUT ####
Query Rewriter Agent created successfully.

--- Testing Query Rewriter Agent ---
Original sub-question: How does AMD 2024 AI chip strategy potentially exacerbat
Rewritten Search Query: analysis of how AMD 2024 AI chip strategy, including pr
```

The original question is for an analyst, he rewritten query is for a search engine. It has been assigned with specific terms like **“MI300X”**, **“market share erosion”** and **“data center”** all of which were synthesized from the keywords and past context.

A query like this is far more likely to retrieve exactly the right documents, making our entire system more accurate and efficient. This rewriting step will be a crucial part of our main agentic loop.

#### Precision with Metadata-Aware Chunking

So, basically, our Planner Agent is giving us a good opportunity. It’s not just saying **find risks**, it’s giving us a hint: **look for risks in the Item 1A. Risk Factors section.**

> But right now, our retriever can’t use that hint. Our vector store is just a big, flat list of 378 text chunks. It has no idea what a “section” is.

Meta aware chunking (Created by Fareed Khan)

We need to fix this. We are going to rebuild our document chunks from scratch. This time, for every single chunk we create, we are going to add a **label** or a **tag** its **metadata** that tells our system exactly which section of the 10-K it came from. This will allow our agent to perform highly precise, filtered searches later on.

First things first, we need a way to programmatically find where each section begins in our raw text file. If we look at the document, we can see a clear pattern: every major section starts with the word **“ITEM”** followed by a number, like **“ITEM 1A”** or **“ITEM 7”**. This is a perfect job for a regular expression.

```
# This regex is designed to find section titles like 'ITEM 1A.' or 'ITEM 7.' in
# It looks for the word 'ITEM', followed by a space, a number, an optional lett
# The `re.IGNORECASE | re.DOTALL` flags make the search case-insensitive and al
section_pattern = r"(ITEM\\s+\\d[A-Z]?\\.\\s*.*?)(?=\\nITEM\\s+\\d[A-Z]?\\.|$)"
```

We are basically creating a pattern that will act as our **section detector**. It should be designed so that it can be flexible enough to catch different formats while being specific enough not to grab the wrong text.

Now we can use this pattern to slice our document into two separate lists: one containing just the section titles, and another containing the content within each section.

```
# We'll work with the raw text loaded earlier from our Document object
raw_text = documents[0].page_content

# Use re.findall to apply our pattern and extract all section titles into a lis
section_titles = re.findall(section_pattern, raw_text, re.IGNORECASE | re.DOTAL

# A quick cleanup step to remove any extra whitespace or newlines from the titl
section_titles = [title.strip().replace('\\n', ' ') for title in section_titles

# Now, use re.split to break the document apart at each point where a section t
sections_content = re.split(section_pattern, raw_text, flags=re.IGNORECASE | re

# The split results in a list with titles and content mixed, so we filter it to
sections_content = [content.strip() for content in sections_content if content.
print(f"Identified {len(section_titles)} document sections.")

# This is a crucial sanity check: if the number of titles doesn't match the num
assert len(section_titles) == len(sections_content), "Mismatch between titles a
```

This is a very effective way to parse a semi-structured document. We have used our regex pattern twice: once to get a clean list of all the section titles, and again to split the main text into a list of content blocks. The `assert` statement gives us confidence that our parsing logic is sound.

Okay, now we have the pieces: a list of titles and a corresponding list of contents. We can now loop through them and create our final, metadata-rich chunks.

```
import uuid # We'll use this to give each chunk a unique ID, which is good prac

# This list will hold our new, metadata-rich document chunks
doc_chunks_with_metadata = []

# Loop through each section's content along with its title using enumerate
for i, content in enumerate(sections_content):
    # Get the corresponding title for the current content block
    section_title = section_titles[i]
    # Use the same text splitter as before, but this time, we run it ONLY on th
    section_chunks = text_splitter.split_text(content)

    # Now, loop through the smaller chunks created from this one section
    for chunk in section_chunks:
        # Generate a unique ID for this specific chunk
        chunk_id = str(uuid.uuid4())
        # Create a new LangChain Document object for the chunk
        doc_chunks_with_metadata.append(
            Document(
                page_content=chunk,
                # This is the most important part: we attach the metadata
                metadata={
                    "section": section_title,      # The section this chunk bel
                    "source_doc": doc_path_clean,  # Where the document came fr
                    "id": chunk_id                 # The unique ID for this chu
                }
            )
        )



print(f"Created {len(doc_chunks_with_metadata)} chunks with section metadata.")
print("\n--- Sample Chunk with Metadata ---")

# To prove it worked, let's find a chunk that we know should be in the 'Risk Fa
sample_chunk = next(c for c in doc_chunks_with_metadata if "Risk Factors" in c.
print(sample_chunk)
```

This is the core of our upgrade. We iterate through each section one by one. For each section, we create our text chunks. But before we add them to our final list, we create a `metadata` dictionary and attach the `section_title`. This effectively **tags** every single chunk with its origin.

Let's look at the output and see the difference.

```
#### OUTPUT ####
Processing document and adding metadata...
Identified 22 document sections.
Created 381 chunks with section metadata.


--- Sample Chunk with Metadata ---
Document(
|   page_content='Our industry is intensely competitive. We operate in the semi
|   metadata={
|   |   'section': 'Item 1A. Risk Factors.',
|   |   'source_doc': './data/nvda_10k_2023_clean.txt',
|   |   'id': '...'
|   }
)
```

Look at that `metadata` block. The same chunk of text we had before now has a piece of context attached: `'section': 'Item 1A. Risk Factors.'`.

Now, when our agent needs to find risks, it can tell the retriever, **"Hey, don't search all 381 chunks. Just search the ones where the section metadata is 'Item 1A. Risk Factors"**.

This simple change transforms our retriever from a blunt instrument into a surgical tool, and it is a key principle for building truly production-grade RAG systems.

### Creating The Multi-Stage Retrieval Funnel

So far, we have engineered a smart planner and enriched our documents with metadata. We are now ready to build the heart of our system: a sophisticated retrieval pipeline.

> A simple, one-shot semantic search is no longer good enough. For a production-grade agent, we need a retrieval process that is both adaptive and multi-stage.

We will design our retrieval process as a funnel, where each stage refines the results of the previous one:

Multi Stage Funnel (Created by Fareed Khan)

- **The Retrieval Supervisor:** We will build a new **supervisor agent** that acts as a dynamic router, analyzing each sub-question and choosing the best search strategy (vector, keyword, or hybrid).
- **Stage 1 (Broad Recall):** We will implement the different retrieval strategies that our supervisor can choose from, focusing on casting a wide net to capture all potentially relevant documents.
- **Stage 2 (High Precision):** We will use a Cross-Encoder model to re-rank the initial results, discarding noise and promoting the most relevant documents to the top.
- **Stage 3 (Synthesis):** Finally, we will create a **Distiller Agent** to compress the top-ranked documents into a single, concise paragraph of context for our downstream agents.

#### Dynamically Choosing a Strategy Using Supervisor

So, basically, not all search queries are the same. A question like “What was the revenue for the ‘Compute & Networking’ segment?” contains specific, exact terms. A keyword-based search would be perfect for that.

But a question like …

> What is the company sentiment on market competition? is conceptual. A semantic, vector-based search would be much better.

Supervisor Agent (Created by Fareed Khan)

Instead of hardcoding one strategy, we are going to build a small, intelligent agent, **Retrieval Supervisor** to make this decision for us. Its only job is to look at the search query and decide which of our retrieval methods is the most appropriate.

First, we need to define the possible decisions our supervisor can make. We'll use a Pydantic `BaseModel` to structure its output.

```
class RetrievalDecision(BaseModel):
    # The chosen retrieval strategy. Must be one of these three options.
    strategy: Literal["vector_search", "keyword_search", "hybrid_search"]
    # The agent's justification for its choice.
    justification: str
```

The supervisor must choose one of these three strategies and explain its reasoning. This makes its decision-making process transparent and reliable.

Now, let's create the prompt that will guide this agent's behavior.

```
retrieval_supervisor_prompt = ChatPromptTemplate.from_messages([
    ("system", """You are a retrieval strategy expert. Based on the user's quer
You have three options:
1. `vector_search`: Best for conceptual, semantic, or similarity-based queries.
2. `keyword_search`: Best for queries with specific, exact terms, names, or cod
3. `hybrid_search`: A good default that combines both, but may be less precise
    ("human", "User Query: {sub_question}") # The rewritten search query will b
])
```

We have created a very direct prompt here which is telling the LLM its role is a **retrieval strategy expert** and clearly explaining when each of its available strategies is most effective.

Finally, we can assemble our supervisor agent.

```
# Create the agent by piping our prompt to the reasoning LLM and structuring it
retrieval_supervisor_agent = retrieval_supervisor_prompt | reasoning_llm.with_s
print("Retrieval Supervisor Agent created.")

# Let's test it with two different types of queries to see how it behaves
print("\n--- Testing Retrieval Supervisor Agent ---")
query1 = "revenue growth for the Compute & Networking segment in fiscal year 20
decision1 = retrieval_supervisor_agent.invoke({"sub_question": query1})

print(f"Query: '{query1}'")
print(f"Decision: {decision1.strategy}, Justification: {decision1.justification

query2 = "general sentiment about market competition and technological innovati
decision2 = retrieval_supervisor_agent.invoke({"sub_question": query2})
print(f"\nQuery: '{query2}'")
print(f"Decision: {decision2.strategy}, Justification: {decision2.justification
```

Here we are wiring it all together.

Our `.with_structured_output(RetrievalDecision)` is again doing the heavy lifting, ensuring we get a clean, predictable `RetrievalDecision` object back from the LLM. Let's look at the test results.

```
#### OUTPUT ####
Retrieval Supervisor Agent created.


# --- Testing Retrieval Supervisor Agent ---
Query: 'revenue growth for the Compute & Networking segment in fiscal year 2023
Decision: keyword_search, Justification: The query contains specific keywords l

Query: 'general sentiment about market competition and technological innovation
Decision: vector_search, Justification: This query is conceptual and seeks to u
```

We can see that it correctly identified that the first query is full of specific terms and chose `keyword_search`.

For the second query, which is conceptual and abstract, it correctly chose `vector_search`. This dynamic decision-making at the start of our retrieval funnel is a good upgrade over a one-size-fits-all approach.

#### Broad Recall with Hybrid, Keyword and Semantic Search

Now that we have a supervisor to choose our strategy, we need to build the retrieval strategies themselves. This first stage of our funnel is all about **Recall** our goal is to cast a wide net and capture every document that could possibly be relevant, even if we pick up some noise along the way.

To do this, we will implement three distinct search functions that our supervisor can call:

1. **Vector Search:** Our standard semantic search, but now upgraded to use metadata filters.
2. **Keyword Search (BM25):** A classic, powerful algorithm that excels at finding documents with specific, exact terms.
3. **Hybrid Search:** A **best of both** the approaches is to combines the results of vector and keyword search using a technique called Reciprocal Rank Fusion (RRF).

First, we need to create a new, advanced vector store using the metadata-enriched chunks we created in the previous section.

```
import numpy as np # A fundamental library for numerical operations in Python
from rank_bm25 import BM25Okapi # The library for implementing the BM25 keyword


print("Creating advanced vector store with metadata...")

# We create a new Chroma vector store, this time using our metadata-rich chunks
advanced_vector_store = Chroma.from_documents(
    documents=doc_chunks_with_metadata,
    embedding=embedding_function
)
print(f"Advanced vector store created with {advanced_vector_store._collection.c
```

This is a simple but critical step. This `advanced_vector_store` now contains the same text as our baseline, but each embedded chunk is tagged with its section title, unlocking our ability to perform filtered searches.

Next, we need to prepare for our keyword search. The BM25 algorithm works by analyzing the frequency of words in documents. To enable this, we need to pre-process our corpus by splitting each document's content into a list of words (tokens).

```
print("\nBuilding BM25 index for keyword search...")

# Create a list where each element is a list of words from a document
tokenized_corpus = [doc.page_content.split(" ") for doc in doc_chunks_with_meta

# Create a list of all unique document IDs
doc_ids = [doc.metadata["id"] for doc in doc_chunks_with_metadata]

# Create a mapping from a document's ID back to the full Document object for ea
doc_map = {doc.metadata["id"]: doc for doc in doc_chunks_with_metadata}

# Initialize the BM25Okapi index with our tokenized corpus
bm25 = BM25Okapi(tokenized_corpus)
```

We are basically creating the necessary data structures for our BM25 index. The `tokenized_corpus` is what the algorithm will search over, and the `doc_map` will allow us to quickly retrieve the full `Document` object after the search is complete.

Now we can define our three retrieval functions.

```
# Strategy 1: Pure Vector Search with Metadata Filtering
def vector_search_only(query: str, section_filter: str = None, k: int = 10):
    # This dictionary defines the metadata filter. ChromaDB will only search do
    filter_dict = {"section": section_filter} if section_filter and "Unknown" n
    # Perform the similarity search with the optional filter
    return advanced_vector_store.similarity_search(query, k=k, filter=filter_di


# Strategy 2: Pure Keyword Search (BM25)
def bm25_search_only(query: str, k: int = 10):
    # Tokenize the incoming query
    tokenized_query = query.split(" ")
    # Get the BM25 scores for the query against all documents in the corpus
    bm25_scores = bm25.get_scores(tokenized_query)
    # Get the indices of the top k documents
    top_k_indices = np.argsort(bm25_scores)[::-1][:k]
    # Use our doc_map to return the full Document objects for the top results
    return [doc_map[doc_ids[i]] for i in top_k_indices]

# Strategy 3: Hybrid Search with Reciprocal Rank Fusion (RRF)
def hybrid_search(query: str, section_filter: str = None, k: int = 10):
    # 1. Perform a keyword search
    bm25_docs = bm25_search_only(query, k=k)
    # 2. Perform a semantic search with the metadata filter
    semantic_docs = vector_search_only(query, section_filter=section_filter, k=
```

```
        # 3. Combine and re-rank the results using Reciprocal Rank Fusion (RRF)
        # Get a unique set of all documents found by either search method
        all_docs = {doc.metadata["id"]: doc for doc in bm25_docs + semantic_docs}.v
        # Create lists of just the document IDs from each search result
        ranked_lists = [[doc.metadata["id"] for doc in bm25_docs], [doc.metadata["i

        # Initialize a dictionary to store the RRF scores for each document
        rrf_scores = {}
        # Loop through each ranked list (BM25 and Semantic)
        for doc_list in ranked_lists:
            # Loop through each document ID in the list with its rank (i)
            for i, doc_id in enumerate(doc_list):
                if doc_id not in rrf_scores:
                    rrf_scores[doc_id] = 0
                # The RRF formula: add 1 / (rank + k) to the score. We use k=61 as
                rrf_scores[doc_id] += 1 / (i + 61)
        # Sort the document IDs based on their final RRF scores in descending order
        sorted_doc_ids = sorted(rrf_scores.keys(), key=lambda x: rrf_scores[x], rev
        # Return the top k Document objects based on the fused ranking
        final_docs = [doc_map[doc_id] for doc_id in sorted_doc_ids[:k]]
        return final_docs
    print("\nAll retrieval strategy functions ready.")
```

We have now implemented the core of our adaptive retrieval system.

- The `vector_search_only` function is our upgraded semantic search. The key addition is the `filter=filter_dict` argument, which allows us to pass the `document_section` from our planner's `Step` and force the search to only consider chunks with that metadata.
- The `bm25_search_only` function is our pure keyword retriever. It's incredibly fast and effective for finding specific terms that semantic search might miss.
- The `hybrid_search` function runs both searches in parallel and then intelligently merges the results using RRF. RRF is a simple but powerful algorithm that ranks documents based on their position in each list, effectively giving more weight to documents that appear high up in *both* search results.

Let's do a quick test to see our keyword search in action. We'll search for the exact section title our planner identified.

```
    # Test Keyword Search to see if it can precisely find a specific section
    print("\n--- Testing Keyword Search ---")
```

```
test_query = "Item 1A. Risk Factors"
test_results = bm25_search_only(test_query)
print(f"Query: {test_query}")
print(f"Found {len(test_results)} documents. Top result section: {test_results[
```

```
#### OUTPUT ####
Creating advanced vector store with metadata...
Advanced vector store created with 381 embeddings.

Building BM25 index for keyword search...
All retrieval strategy functions ready.

# --- Testing Keyword Search ---
Query: Item 1A. Risk Factors
Found 10 documents. Top result section: Item 1A. Risk Factors.
```

The output is exactly what we wanted. The BM25 search, being keyword-focused, was able to perfectly and instantly retrieve the documents from the **Item 1A. Risk Factors** section, just by searching for the title.

> Our supervisor can now choose this precise tool when the query contains specific keywords like a section title.

With our broad recall stage now built, we have a powerful mechanism for finding all potentially relevant documents. However, this wide net can also bring in irrelevant noise. The next stage of our funnel will focus on filtering this down with high precision.

#### High Precision Using a Cross-Encoder Reranker

So, our Stage 1 retrieval is doing a great job at **Recall.** It's pulling in 10 documents that are potentially relevant to our sub-question.

> But that's the problem they are only potentially relevant. Feeding all 10 of these chunks directly to our main reasoning LLM is inefficient and risky.

**It increases token costs and, more importantly, it can confuse the model with noisy, semi-relevant information.**

High Precision (Created by Fareed Khan)

What we need now is a **Precision stage.** We need a way to inspect those 10 candidate documents and pick out the absolute best ones. This is where a **Reranker** comes in.

The key difference is how these models work.

1. Our initial retrieval uses a **bi-encoder** (the embedding model), which creates a vector for the query and documents independently. It's fast and great for searching over millions of items.
2. A **cross-encoder**, on the other hand, takes the query and a single document together as a pair and performs a much deeper, more nuanced comparison. It's slower, but far more accurate.

So, basically, we want to build a function that takes our 10 retrieved documents and uses a cross-encoder model to give each one a precise relevance score. Then, we will keep only the top 3, as defined in our `config`.

First, let's initialize our cross-encoder model. We'll use a small but highly effective model from the `sentence-transformers` library, as specified in our configuration.

```
from sentence_transformers import CrossEncoder # The library for using cross-en

print("Initializing CrossEncoder reranker...")

# Initialize the CrossEncoder model using the name from our central config dict
# The library will automatically download the model from the Hugging Face Hub i
reranker = CrossEncoder(config["reranker_model"])
```

We are basically loading the pre-trained reranking model into memory. This only needs to be done once. The model we have chosen, `ms-marco-MiniLM-L-6-v2`, is very popular for this task because it offers a great balance of speed and accuracy.

Now we can create the function that will perform the reranking.

```
def rerank_documents_function(query: str, documents: List[Document]) -> List[Do
    # If we have no documents to rerank, return an empty list immediately.
    if not documents:
        return []

    # Create the pairs of [query, document_content] that the cross-encoder need
    pairs = [(query, doc.page_content) for doc in documents]

    # Use the reranker to predict a relevance score for each pair. This returns
    scores = reranker.predict(pairs)

    # Combine the original documents with their new scores.
    doc_scores = list(zip(documents, scores))

    # Sort the list of (document, score) tuples in descending order based on th
    doc_scores.sort(key=lambda x: x[1], reverse=True)

    # Extract just the Document objects from the top N sorted results.
    # The number of documents to keep is controlled by 'top_n_rerank' in our co
    reranked_docs = [doc for doc, score in doc_scores[:config["top_n_rerank"]]]
```

```
    return reranked_docs
```

This function, `rerank_documents_function`, is the main part of our precision stage. It takes the `query` and the list of 10 `documents` from our recall stage. The most important step is `reranker.predict(pairs)`.

Here, the model isn't creating embeddings, it's performing a full comparison of the query against each document content, producing a relevance score for each one.

After getting the scores, we simply sort the documents and slice the list to keep only the top 3. The output of this function will be a short, clean, and highly relevant list of documents the perfect context for our downstream agents.

This funneling approach, moving from a high-recall first stage to a high-precision second stage, is a component of a production-grade RAG system. **It ensures we get the best possible evidence while minimizing noise and cost.**

#### Synthesizing using Contextual Distillation

So, our retrieval funnel has worked beautifully. We started with a broad search that gave us 10 potentially relevant documents. Then, our high-precision reranker filtered that down to the top 3, most relevant chunks.

We are in a much better position now, but we can still make one final improvement before handing this information over to our main reasoning agents. Right now, we have three separate text chunks.

While they are all relevant, they might contain redundant information or overlapping sentences. Presenting them as three distinct blocks can still be a bit clunky for an LLM to process.

The final stage of our retrieval funnel is **Contextual Distillation**. The goal is simple: take our top 3 highly relevant document chunks and distill them into a single, clean, and concise paragraph. This removes any final redundancy and presents a perfectly synthesized piece of evidence to our downstream agents.

> This distillation step acts as a final compression layer. It ensures the context fed into our more expensive reasoning agents is as dense and information-rich as possible, maximizing signal and minimizing noise.

To do this, we will create another small, specialized agent that we will call the **Distiller Agent**.

First, we need to design the prompt that will guide its behavior.

```
# The prompt for our distiller agent, instructing it to synthesize and be conci
distiller_prompt = ChatPromptTemplate.from_messages([
    ("system", """You are a helpful assistant. Your task is to synthesize the f
The goal is to provide a clear and coherent context that directly answers the q
Focus on removing redundant information and organizing the content logically. A
    ("human", "Retrieved Documents:\n{context}") # The content of our top 3 rer
])
```

We are basically giving this agent a very focused task. We're telling it: **"Here are some pieces of text. Your only job is to merge them into one coherent paragraph that answers this specific question"**. The instruction to **"Answer only with the synthesized context"** is important, it prevents the agent from adding any conversational fluff or trying to answer the question itself. It's purely a text-processing tool.

Now, we can assemble our simple `distiller_agent`.

```
# Create the agent by piping our prompt to the reasoning LLM and a string outpu
distiller_agent = distiller_prompt | reasoning_llm | StrOutputParser()
print("Contextual Distiller Agent created.")
```

This is another straightforward LCEL chain. We take our `distiller_prompt`, pipe it to our powerful `reasoning_llm` to perform the synthesis, and then use a `StrOutputParser` to get the final, clean paragraph of text.

With this `distiller_agent` created, our multi-stage retrieval funnel is now complete. In our main agentic loop, the flow for each research step will be:

1. **Supervisor:** Choose a retrieval strategy (`vector`, `keyword`, or `hybrid`).
2. **Recall Stage:** Execute the chosen strategy to get the top 10 documents.
3. **Precision Stage:** Use the `rerank_documents_function` to get the top 3 documents.
4. **Distillation Stage:** Use the `distiller_agent` to compress the top 3 documents into a single, clean paragraph.

This multi-stage process ensures that the evidence our agent works with is of the highest possible quality. The next step is to give our agent the ability to look beyond its internal knowledge and search the web.

### Augmenting Knowledge with Web Search

So, our retrieval funnel is now incredibly powerful but it has one massive blind spot.

> It can only see what's inside our 2023 10-K document. To solve our challenge query, our agent needs to find recent news (post-filing, from 2024) about AMD's AI chip strategy. That information simply does not exist in our static knowledge base.

To truly build a “**Deep Thinking**” agent, it needs to be able to recognize the limits of its own knowledge and look for answers elsewhere. We need to give it a window to the outside world.

Augemtation using Web (Created by Fareed Khan)

This is the step where we augment our agent’s capabilities with a new tool: **Web Search**. This transforms our system from a document-specific Q&A bot into a true, multi-source research assistant.

For this, we will use the **Tavily Search API**. It’s a search engine built specifically for LLMs, providing clean, ad-free, and relevant search results that are perfect for RAG pipelines. It also integrates seamlessly with LangChain.

So, basically, the first thing we need to do is initialize the Tavily search tool itself.

```
from langchain_community.tools.tavily_search import TavilySearchResults

# Initialize the Tavily search tool.
# k=3: This parameter instructs the tool to return the top 3 most relevant sear
web_search_tool = TavilySearchResults(k=3)
```

We are basically creating an instance of the Tavily search tool that our agent can call. The `k=3` parameter is a good starting point, providing a few high-quality sources

without overwhelming the agent with too much information.

Now, a raw API response isn't quite what we need. Our downstream components, the reranker and the distiller are all designed to work with a specific data structure: a list of LangChain `Document` objects. To ensure seamless integration, we need to create a simple wrapper function. This function will take a query, call the Tavily tool, and then format the raw results into that standard `Document` structure.

```
def web_search_function(query: str) -> List[Document]:
    # Invoke the Tavily search tool with the provided query.
    results = web_search_tool.invoke({"query": query})

    # Format the results into a list of LangChain Document objects.
    # We use a list comprehension for a concise and readable implementation.
    return [
        Document(
            # The main content of the search result goes into 'page_content'.
            page_content=res["content"],
            # We store the source URL in the 'metadata' dictionary for citation
            metadata={"source": res["url"]}
        ) for res in results
    ]
```

This `web_search_function` acts as a crucial adapter. It calls `web_search_tool.invoke` which returns a list of dictionaries, with each dictionary containing keys like `"content"` and `"url"`.

1. The list comprehension then loops through these results and neatly repackages them into the `Document` objects our pipeline expects.
2. The `page_content` gets the main text, and importantly, we store the `url` in the `metadata`.
3. This ensures that when our agent generates its final answer, it can properly cite its web sources.

This makes our external knowledge source look and feel exactly like our internal one, allowing us to use the same processing pipeline for both.

With our function ready, let's give it a quick test to make sure it's working as expected. We'll use a query that's relevant to the second part of our main challenge.

```
# Test the web search function with a query about AMD's 2024 strategy
print("\n--- Testing Web Search Tool ---")
test_query_web = "AMD AI chip strategy 2024"
test_results_web = web_search_function(test_query_web)
print(f"Found {len(test_results_web)} results for query: '{test_query_web}'")
# Print a snippet from the first result to see what we got back
if test_results_web:
    print(f"Top result snippet: {test_results_web[0].page_content[:250]}...")
```

```
#### OUTPUT ####
Web search tool (Tavily) initialized.

--- Testing Web Search Tool ---
Found 3 results for query: 'AMD AI chip strategy 2024'
Top result snippet: AMD has intensified its battle with Nvidia in the AI chip m
```

The output confirms that our tool is working perfectly. It found 3 relevant web pages for our query. The snippet from the top result is exactly the kind of up-to-date, external information our agent was missing.

> It mentions AMD "Instinct MI300X" and its competition with NVIDIA "H100" precisely the evidence needed to solve the second half of our problem.

Our agent now has a window to the outside world, and its planner can intelligently decide when to look through it. The final piece of the puzzle is to give the agent the ability to reflect on its findings and decide when its research is complete.

### Self-Critique and Control Flow Policy

So far, we have built a powerful research machine. Our agent can create a plan, choose the right tools, and execute a sophisticated retrieval funnel. But one critical

piece is missing: the ability to **think about its own progress.** An agent that blindly follows a plan, step by step, is not truly intelligent. It needs a mechanism for self-critique.

Self Critique and Policy Making (Created by Fareed Khan)

This is where we build the cognitive core of our agent autonomy. After each research step, our agent will pause and reflect. It will look at the new information it just found, compare it to what it already knew, and then make a strategic decision: is my research complete, or do I need to continue?

> This self-critique loop is what elevates our system from a scripted workflow to an autonomous agent. It's the mechanism that allows it to decide when it has gathered enough evidence to confidently answer the user's question.

We will implement this using two new specialized agents:

1. **The Reflection Agent:** This agent will take the distilled context from a completed step and create a concise, one-sentence summary. This summary is then added

to our agent's "research history."

2. **The Policy Agent:** This is the master strategist. After reflection, it will examine the *entire* research history in relation to the original plan and make a crucial decision: `CONTINUE_PLAN` or `FINISH`.

#### Update and Reflect Cumulative Research History

After our agent completes a research step (e.g., retrieving and distilling information about NVIDIA's risks), we don't want to just move on. We need to integrate this new knowledge into the agent's memory.

Reflective Cumulative (Created by Fareed Khan)

We will build a **Reflection Agent** whose only job is to perform this integration. It will take the rich, distilled context from the current step and summarize it into a single, factual sentence. This summary then gets added to the `past_steps` list in our `RAGState`.

First, let's create the prompt for this agent.

```
# The prompt for our reflection agent, instructing it to be concise and factual
reflection_prompt = ChatPromptTemplate.from_messages([
    ("system", """You are a research assistant. Based on the retrieved context
This summary will be added to our research history. Be factual and to the point
    ("human", "Current sub-question: {sub_question}\n\nDistilled context:\n{con
])
```

We are telling this agent to act like a diligent research assistant. Its task is not to be creative, but to be a good note-taker. It reads the `context` and writes a `summary`. Now we can assemble the agent itself.

```
# Create the agent by piping our prompt to the reasoning LLM and a string outpu
reflection_agent = reflection_prompt | reasoning_llm | StrOutputParser()
print("Reflection Agent created.")
```

This `reflection_agent` is a part of our cognitive loop. By creating these concise summaries, it builds up a clean, easy-to-read **research history**. This history will be the input for our next, and most important, agent: the one that decides when to stop.

#### Building Policy Agent for Control Flow

This is the brain of our agent autonomy. After the `reflection_agent` has updated the research history, the **Policy Agent** comes into play. It acts as the supervisor of the whole operation.

Its job is to look at everything the agent knows — the original question, the initial plan, and the full history of summaries from completed steps and make a high-level strategic decision.

We will start by defining the structure of its decision using a Pydantic model.

```
class Decision(BaseModel):
    # The decision must be one of these two actions.
    next_action: Literal["CONTINUE_PLAN", "FINISH"]
    # The agent must justify its decision.
    justification: str
```

This `Decision` class forces our Policy Agent to make a clear, binary choice and to explain its reasoning. This makes its behavior transparent and easy to debug.

Next, we design the prompt that will guide its decision-making process.

```
# The prompt for our policy agent, instructing it to act as a master strategist
policy_prompt = ChatPromptTemplate.from_messages([
    ("system", """You are a master strategist. Your role is to analyze the rese
You have the original question, the initial plan, and a log of completed steps
- If the collected information in the Research History is sufficient to compreh
- Otherwise, if the plan is not yet complete, decide to CONTINUE_PLAN."""),
    ("human", "Original Question: {question}\n\nInitial Plan:\n{plan}\n\nResear
])
```

We are basically asking the LLM to perform a meta-analysis. It's not answering the question itself; it's reasoning about the *state of the research process*. It compares what it has (`history`) with what it needs (`plan` and `question`) and makes a judgment call.

Now, we can assemble the `policy_agent`.

```
# Create the agent by piping our prompt to the reasoning LLM and structuring it
policy_agent = policy_prompt | reasoning_llm.with_structured_output(Decision)
print("Policy Agent created.")

# Now, let's test the policy agent with two different states of our research pr
print("\n--- Testing Policy Agent (Incomplete State) ---")

# First, a state where only Step 1 is complete.
plan_str = json.dumps([s.dict() for s in test_plan.steps])
incomplete_history = "Step 1 Summary: NVIDIA's 10-K states that the semiconduct
```

```
decision1 = policy_agent.invoke({"question": complex_query_adv, "plan": plan_st
print(f"Decision: {decision1.next_action}, Justification: {decision1.justificat
print("\n--- Testing Policy Agent (Complete State) ---")

# Second, a state where both Step 1 and Step 2 are complete.
complete_history = incomplete_history + "\nStep 2 Summary: In 2024, AMD launche
decision2 = policy_agent.invoke({"question": complex_query_adv, "plan": plan_st
print(f"Decision: {decision2.next_action}, Justification: {decision2.justificat
```

To properly test our `policy_agent`, we simulate two distinct moments in our agent's lifecycle. In the first test, we provide it with a history that only contains the summary from Step 1. In the second, we provide it with the summaries from both Step 1 and Step 2.

Let's examine its decisions in each case.

```
#### OUTPUT ####
Policy Agent created.

--- Testing Policy Agent (Incomplete State) ---
Decision: CONTINUE_PLAN, Justification: The research has only identified NVIDIA

--- Testing Policy Agent (Complete State) ---
Decision: FINISH, Justification: The research history now contains comprehensiv
```

Let's understand the output ...

- **In the incomplete state,** the agent correctly recognized that it was missing the information about AMD's strategy. It looked at its plan, saw that the next step was to use the web search, and correctly decided to `CONTINUE_PLAN`.
- **In the complete state,** after being given the summary from the web search, it analyzed its history again. This time, it recognized that it had all the pieces of the puzzle NVIDIA risks and AMD strategy. It correctly decided that its research was done and it was time to `FINISH`.

With this `policy_agent`, we have built the brain of our autonomous system. The final step is to wire all of these components together into a complete, executable workflow using LangGraph.

## Defining the Graph Nodes

we have designed all these cool, specialized agents. Now it's time to turn them into the actual building blocks of our workflow. In LangGraph, these building blocks are called **nodes**. A node is just a Python function that does one specific job. It takes the agent's current memory (`RAGState`) as input, performs its task, and then returns a dictionary with any updates to that memory.

We will create a node for every major step our agent needs to take.

Graph Nodes (Created by Fareed Khan)

First up, we need a simple helper function. Since our agents will often need to see the research history, we want a clean way to format the `past_steps` list into a readable string.

```
# A helper function to format the research history for prompts
def get_past_context_str(past_steps: List[PastStep]) -> str:
    # This takes the list of PastStep dictionaries and joins them into a single
    # Each step is clearly labeled for the LLM to understand the context.
    return "\\n\\n".join([f"Step {s['step_index']}: {s['sub_question']}\\nSumma
```

We are basically creating a utility that will be used inside several of our nodes to provide historical context to our prompts.

Now for our first real node: the `plan_node`. This is the starting point of our agent's reasoning. Its only job is to call our `planner_agent` and populate the `plan` field in our `RAGState`.

```
# Node 1: The Planner
def plan_node(state: RAGState) -> Dict:
    console.print("--- 🧠: Generating Plan ---")
    # We call the planner_agent we created earlier, passing in the user's origi
    plan = planner_agent.invoke({"question": state["original_question"]})
    rprint(plan)
    # We return a dictionary with the updates for our RAGState.
```

```
    # LangGraph will automatically merge this into the main state.
    return {"plan": plan, "current_step_index": 0, "past_steps": []}
```

This node kicks everything off. It takes the `original_question` from the state, gets the `plan`, and then initializes the `current_step_index` to 0 (to start with the first step) and clears the `past_steps` history for this new run.

Next, we need the nodes that actually go and find information. Since our planner can choose between two tools, we need two separate retrieval nodes. Let's start with the `retrieval_node` for searching our internal 10-K document.

```
# Node 2a: Retrieval from the 10-K document
def retrieval_node(state: RAGState) -> Dict:
    # First, get the details for the current step in the plan.
    current_step_index = state["current_step_index"]
    current_step = state["plan"].steps[current_step_index]
    console.print(f"--- 🔍: Retrieving from 10-K (Step {current_step_index + 1}

    # Use our query rewriter to optimize the sub-question for search.
    past_context = get_past_context_str(state['past_steps'])
    rewritten_query = query_rewriter_agent.invoke({
        "sub_question": current_step.sub_question,
        "keywords": current_step.keywords,
        "past_context": past_context
    })
    console.print(f"  Rewritten Query: {rewritten_query}")

    # Get the supervisor's decision on which retrieval strategy is best.
    retrieval_decision = retrieval_supervisor_agent.invoke({"sub_question": rew
    console.print(f"  Supervisor Decision: Use `{retrieval_decision.strategy}`.

    # Based on the decision, execute the correct retrieval function.
    if retrieval_decision.strategy == 'vector_search':
        retrieved_docs = vector_search_only(rewritten_query, section_filter=cur
    elif retrieval_decision.strategy == 'keyword_search':
        retrieved_docs = bm25_search_only(rewritten_query, k=config['top_k_retr
    else: # hybrid_search
        retrieved_docs = hybrid_search(rewritten_query, section_filter=current_

    # Return the retrieved documents to be added to the state.
    return {"retrieved_docs": retrieved_docs}
```

This node is doing a lot of intelligent work. It's not just a simple retriever. It orchestrates a mini-pipeline: it rewrites the query, asks the supervisor for the best strategy, and then executes that strategy.

Now, we need the corresponding node for our other tool: web search.

```
# Node 2b: Retrieval from the Web
def web_search_node(state: RAGState) -> Dict:
    # Get the details for the current step.
    current_step_index = state["current_step_index"]
    current_step = state["plan"].steps[current_step_index]
    console.print(f"--- 🌐 : Searching Web (Step {current_step_index + 1}: {curr

    # Rewrite the sub-question for a web search engine.
    past_context = get_past_context_str(state['past_steps'])
    rewritten_query = query_rewriter_agent.invoke({
        "sub_question": current_step.sub_question,
        "keywords": current_step.keywords,
        "past_context": past_context
    })
    console.print(f"  Rewritten Query: {rewritten_query}")
    # Call our web search function.
    retrieved_docs = web_search_function(rewritten_query)
    # Return the results.
    return {"retrieved_docs": retrieved_docs}
```

This `web_search_node` is simpler because it doesn't need a supervisor, it just has one way to search the web. But it still uses our powerful query rewriter to make sure the search is as effective as possible.

After we retrieve documents (from either source), we need to run our precision and synthesis funnel. We'll create a node for each stage. First, the `rerank_node`.

```
# Node 3: The Reranker
def rerank_node(state: RAGState) -> Dict:
    console.print("--- 🎯 : Reranking Documents ---")
    # Get the current step's details.
    current_step_index = state["current_step_index"]
    current_step = state["plan"].steps[current_step_index]
    # Call our reranking function on the documents we just retrieved.
    reranked_docs = rerank_documents_function(current_step.sub_question, state[
    console.print(f"  Reranked to top {len(reranked_docs)} documents.")
```

```
    # Update the state with the high-precision documents.
    return {"reranked_docs": reranked_docs}
```

This node takes the `retrieved_docs` (our broad recall of 10 documents) and uses the cross-encoder to filter them down to the top 3, placing the result in `reranked_docs`.

Next, the `compression_node` will take those top 3 documents and distill them.

```
# Node 4: The Compressor / Distiller
def compression_node(state: RAGState) -> Dict:
    console.print("--- ✂️: Distilling Context ---")
    # Get the current step's details.
    current_step_index = state["current_step_index"]
    current_step = state["plan"].steps[current_step_index]
    # Format the top 3 documents into a single string.
    context = format_docs(state["reranked_docs"])
    # Call our distiller agent to synthesize them into one paragraph.
    synthesized_context = distiller_agent.invoke({"question": current_step.sub_
    console.print(f"  Distilled Context Snippet: {synthesized_context[:200]}...
    # Update the state with the final, clean context.
    return {"synthesized_context": synthesized_context}
```

This node is the last step of our retrieval funnel. It takes the `reranked_docs` and produces a single, clean `synthesized_context` paragraph.

Now that we have our evidence, we need to reflect on it and update our research history. This is the job of the `reflection_node`.

```
# Node 5: The Reflection / Update Step
def reflection_node(state: RAGState) -> Dict:
    console.print("--- : Reflecting on Findings ---")
    # Get the current step's details.
    current_step_index = state["current_step_index"]
    current_step = state["plan"].steps[current_step_index]
    # Call our reflection agent to summarize the findings.
    summary = reflection_agent.invoke({"sub_question": current_step.sub_questio
    console.print(f"  Summary: {summary}")

    # Create a new PastStep dictionary with all the results from this step.
    new_past_step = {
```

```
            "step_index": current_step_index + 1,
            "sub_question": current_step.sub_question,
            "retrieved_docs": state['reranked_docs'], # We save the reranked docs f
            "summary": summary
        }
        # Append the new step to our history and increment the step index to move t
        return {"past_steps": state["past_steps"] + [new_past_step], "current_step_
```

This node is the bookkeeper of our agent. It calls the `reflection_agent` to create the summary and then neatly packages all the results of the current research cycle into a `new_past_step` object. It then adds this to the `past_steps` list and increments the `current_step_index`, getting the agent ready for the next loop.

Finally, when the research is complete, we need one last node to generate the final answer.

```
# Node 6: The Final Answer Generator
def final_answer_node(state: RAGState) -> Dict:
    console.print("--- ✅: Generating Final Answer with Citations ---")
    # First, we need to gather all the evidence we've collected from ALL past s
    final_context = ""
    for i, step in enumerate(state['past_steps']):
        final_context += f"\\n--- Findings from Research Step {i+1} ---\\n"
        # We include the source metadata (section or URL) for each document to
        for doc in step['retrieved_docs']:
            source = doc.metadata.get('section') or doc.metadata.get('source')
            final_context += f"Source: {source}\\nContent: {doc.page_content}\\

    # We create a new prompt specifically for generating the final, citable ans
    final_answer_prompt = ChatPromptTemplate.from_messages([
        ("system", """You are an expert financial analyst. Synthesize the resea
Your answer must be grounded in the provided context. At the end of any sentenc
        ("human", "Original Question: {question}\n\nResearch History and Contex
    ])

    # We create a temporary agent for this final task and invoke it.
    final_answer_agent = final_answer_prompt | reasoning_llm | StrOutputParser(
    final_answer = final_answer_agent.invoke({"question": state['original_quest
    # Update the state with the final answer.
    return {"final_answer": final_answer}
```

This `final_answer_node` is our grand finale. It consolidates all the high-quality, reranked documents from every step in the `past_steps` history into one massive context. It then uses a dedicated prompt to instruct our powerful `reasoning_llm` to synthesize this information into a comprehensive, multi-paragraph answer that includes citations, bringing our research process to a successful conclusion.

With all our nodes defined, we now have all the building blocks for our agent. The next step is to define the "wires" that connect them and control the flow of the graph.

### Defining the Conditional Edges

So, we have built all of our nodes. We have a planner, retrievers, a reranker, a distiller, and a reflector. Think of them as a collection of experts in a room. Now we need to define the rules of conversation. Who speaks when? How do we decide what to do next?

This is the job of **edges** in LangGraph. Simple edges are straightforward, **"after Node A, always go to Node B"**. But the real intelligence comes from **conditional edges**.

> A conditional edge is a function that looks at the agent's current memory (`RAGState`) and makes a decision, routing the workflow down different paths based on the situation.

We need two key decision-making functions for our agent:

1. **A Tool Router (`route_by_tool`):** After the plan is made, this function will look at the *current step* of the plan and decide whether to send the workflow to the `retrieve_10k` node or the `retrieve_web` node.
2. **The Main Control Loop (`should_continue_node`):** This is the most important one. After each research step is completed and reflected upon, this function will call our `policy_agent` to decide whether to continue to the next step in the plan or to finish the research and generate the final answer.

First, let's build our simple tool router.

```
# Conditional Edge 1: The Tool Router
def route_by_tool(state: RAGState) -> str:
    # Get the index of the current step we are on.
    current_step_index = state["current_step_index"]
    # Get the full details of the current step from the plan.
    current_step = state["plan"].steps[current_step_index]
    # Return the name of the tool specified for this step.
    # LangGraph will use this string to decide which node to go to next.
    return current_step.tool
```

This function is very simple, but crucial. It acts as a switchboard. It reads the `current_step_index` from the state, finds the corresponding `Step` in the `plan`, and returns the value of its `tool` field (which will be either `"search_10k"` or `"search_web"`). When we wire up our graph, we will tell it to use this function's output to choose the next node.

Now we need to create a function that controls our agent's primary reasoning loop. This is where our `policy_agent` comes into play.

```
# Conditional Edge 2: The Main Control Loop
def should_continue_node(state: RAGState) -> str:
    console.print("--- 🚦 : Evaluating Policy ---")
    # Get the index of the step we are about to start.
    current_step_index = state["current_step_index"]

    # First, check our basic stopping conditions.
    # Condition 1: Have we completed all the steps in the plan?
    if current_step_index >= len(state["plan"].steps):
        console.print("  -> Plan complete. Finishing.")
        return "finish"

    # Condition 2: Have we exceeded our safety limit for the number of iteratio
    if current_step_index >= config["max_reasoning_iterations"]:
        console.print("  -> Max iterations reached. Finishing.")
        return "finish"

    # A special case: If the last retrieval step failed to find any documents,
    # there's no point in reflecting. It's better to just move on to the next s
    if state.get("reranked_docs") is not None and not state["reranked_docs"]:
        console.print("  -> Retrieval failed for the last step. Continuing with
        return "continue"

    # If none of the basic conditions are met, it's time to ask our Policy Agen
    # We format the history and plan into strings for the prompt.
```

```
        history = get_past_context_str(state['past_steps'])
        plan_str = json.dumps([s.dict() for s in state['plan'].steps])

        # Invoke the policy agent to get its strategic decision.
        decision = policy_agent.invoke({"question": state["original_question"], "pl
        console.print(f"  -> Decision: {decision.next_action} | Justification: {dec

        # Based on the agent's decision, return the appropriate signal.
        if decision.next_action == "FINISH":
            return "finish"
        else: # CONTINUE_PLAN
            return "continue"
```

This `should_continue_node` function is the cognitive core of our agent's control flow. It runs after every `reflection_node`.

1. It first checks for simple, hardcoded **stopping criteria**. Has the plan run out of steps? Have we hit our `max_reasoning_iterations` safety limit? These prevent the agent from running forever.
2. If those checks pass, it then invokes our powerful `policy_agent`. It gives the policy agent all the context it needs: the original goal (`question`), the full `plan`, and the `history` of what's been accomplished so far.
3. Finally, it takes the `policy_agent`'s structured output (`CONTINUE_PLAN` or `FINISH`) and returns the simple string `"continue"` or `"finish"`. LangGraph will use this string to either loop back for another research cycle or proceed to the `final_answer_node`.

With our nodes (the experts) and our conditional edges (the rules of conversation) now defined, we have everything we need.

> It's time to assemble all these pieces into a complete, functioning `StateGraph`.

### Wiring the Deep Thinking RAG Machine

We have all of our individual components ready to go:

1. our nodes (**workers**)
2. our conditional edges (**managers**).

Now it's time to wire them all together into a single, cohesive system.

We will use LangGraph's `StateGraph` to define the complete cognitive architecture of our agent. This is where we lay out the blueprint of our agent's thought process, defining exactly how information flows from one step to the next.

The first thing we need to do is create an instance of the `StateGraph`. We will tell it that the **"state"** it will be passing around is our `RAGState` dictionary.

```
from langgraph.graph import StateGraph, END # Import the main graph components

# Instantiate the graph, telling it to use our RAGState TypedDict as its state
graph = StateGraph(RAGState)
```

We now have an empty graph. The next step is to add all the nodes we defined earlier. The `.add_node()` method takes two arguments: a unique string name for the node, and the Python function that the node will execute.

```
# Add all of our Python functions as nodes in the graph
graph.add_node("plan", plan_node)                       # The node that creates t
graph.add_node("retrieve_10k", retrieval_node)          # The node for internal d
graph.add_node("retrieve_web", web_search_node)         # The node for external w
graph.add_node("rerank", rerank_node)                   # The node that performs
graph.add_node("compress", compression_node)            # The node that distills
graph.add_node("reflect", reflection_node)              # The node that summarize
graph.add_node("generate_final_answer", final_answer_node) # The node that synt
```

Now all our experts are in the room. The final and most critical step is to define the "wires" that connect them. This is where we use the `.add_edge()` and `.add_conditional_edges()` methods to define the flow of control.

```
# The entry point of our graph is the "plan" node. Every run starts here.
graph.set_entry_point("plan")

# After the "plan" node, we use our first conditional edge to decide which tool
graph.add_conditional_edges(
```

```
    "plan",           # The source node
    route_by_tool,    # The function that makes the decision
    {                 # A dictionary mapping the function's output string to th
        "search_10k": "retrieve_10k",
        "search_web": "retrieve_web",
    },
)

# After retrieving from either the 10-K or the web, the flow is linear for a bi
graph.add_edge("retrieve_10k", "rerank") # After internal retrieval, always go
graph.add_edge("retrieve_web", "rerank") # After web retrieval, also always go
graph.add_edge("rerank", "compress")      # After reranking, always go to compr
graph.add_edge("compress", "reflect")     # After compressing, always go to ref

# After the "reflect" node, we hit our main conditional edge, which controls th
graph.add_conditional_edges(
    "reflect",        # The source node
    should_continue_node, # The function that calls our Policy Agent
    {                 # A dictionary mapping the decision to the next step
        "continue": "plan", # If the decision is "continue", we loop back to th
        "finish": "generate_final_answer", # If the decision is "finish", we pr
    },
)

# The "generate_final_answer" node is the last step before the end.
graph.add_edge("generate_final_answer", END) # After generating the answer, the
print("StateGraph constructed successfully.")
```

This is the blueprint of our agent's brain. Let's trace the flow:

1. It always starts at `plan`.
2. The `route_by_tool` conditional edge then acts as a switch, directing the flow to either `retrieve_10k` or `retrieve_web`.
3. Regardless of which retriever runs, the output is always funneled through the `rerank` -> `compress` -> `reflect` pipeline.
4. This brings us to the most important part: the `should_continue_node` conditional edge. This is the heart of our cyclical reasoning.

- If the policy agent says `CONTINUE_PLAN`, the edge sends the workflow all the way back to the `plan` node. We go back to `plan` (instead of directly to the next retriever) so that `route_by_tool` can correctly route the *next* step in the plan.

- If the policy agent says `FINISH`, the edge breaks the loop and sends the workflow to the `generate_final_answer` node.
- Finally, after the answer is generated, the graph terminates at `END`.

We have successfully defined the complete, complex, and cyclical architecture of our Deep Thinking Agent. The only thing left to do is to compile this blueprint into a runnable application and visualize it to see what we have built.

### Compiling and Visualizing the Iterative Workflow

With our graph fully wired, the final step in the assembly process is to **compile** it. The `.compile()` method takes our abstract definition of nodes and edges and turns it into a concrete, executable application.

We can then use a built-in LangGraph utility to generate a diagram of our graph. Visualizing the workflow is incredibly helpful for understanding and debugging complex agentic systems. It transforms our code into an intuitive flowchart that clearly shows the agent's possible reasoning paths.

So, basically, we're taking our blueprint and turning it into a real machine.

```
# The .compile() method takes our graph definition and creates a runnable objec
deep_thinking_rag_graph = graph.compile()
print("Graph compiled successfully.")


# Now, let's visualize the architecture we've built.
try:
    from IPython.display import Image, display
    # We can get a PNG image of the graph's structure.
    png_image = deep_thinking_rag_graph.get_graph().draw_png()
    # Display the image in our notebook.
    display(Image(png_image))
except Exception as e:
    # This can fail if pygraphviz and its system dependencies are not installed
    print(f"Graph visualization failed: {e}. Please ensure pygraphviz is instal
```

The `deep_thinking_rag_graph` object is now our fully functional agent. The visualization code then calls `.get_graph().draw_png()` to generate a visual representation of the state machine we have constructed.

Deep Thinking Simpler Pipeline Flow (Created by Fareed Khan)

We can clearly see:

- The initial branching logic where `route_by_tool` chooses between `retrieve_10k` and `retrieve_web`.

- The linear processing pipeline for each research step (`rerank` -> `compress` -> `reflect`).
- The crucial **feedback loop** where the `should_continue` edge sends the workflow back to the `plan` node to begin the next research cycle.
- The final "exit ramp" that leads to `generate_final_answer` once the research is complete.

This is the architecture of a system that can think. Now, let's put it to the test.

### Running the Deep Thinking Pipeline

We have engineered a reasoning engine. Now it's time to see if it can succeed where our baseline system so spectacularly failed.

We will invoke our compiled `deep_thinking_rag_graph` with the exact same multi-hop, multi-source challenge query. We will use the `.stream()` method to get a real-time, step-by-step trace of the agent's execution, observing its "thought process" as it works through the problem.

Here's the plan for this section:

- **Invoke the Graph:** We'll run our agent and watch as it executes its plan, switching between tools and building its research history.
- **Analyze the Final Output:** We'll examine the final, synthesized answer to see if it successfully integrated information from both the 10-K and the web.
- **Compare the Results:** We will do a final side-by-side comparison to definitively highlight the architectural advantages of our Deep Thinking agent.

We will set up our initial input, which is just a dictionary containing the `original_question`, and then call the `.stream()` method. The `stream` method is fantastic for debugging and observation because it yields the state of the graph after each and every node completes its work.

```
# This will hold the final state of the graph after the run is complete.
final_state = None
# The initial input for our graph, containing the original user query.
graph_input = {"original_question": complex_query_adv}
```

```
print("--- Invoking Deep Thinking RAG Graph ---")
# We use .stream() to watch the agent's process in real-time.
# "values" mode means we get the full RAGState object after each step.
for chunk in deep_thinking_rag_graph.stream(graph_input, stream_mode="values"):
    # The final chunk in the stream will be the terminal state of the graph.
    final_state = chunk
print("\n--- Graph Stream Finished ---")
```

This loop is where our agent comes to life. With each iteration, LangGraph executes the next node in the workflow, updates the `RAGState`, and yields the new state to us. The `rich` library `console.print` statements that we embedded inside our nodes will give us a running commentary of the agent's actions and decisions.

```
#### OUTPUT ####

--- Invoking Deep Thinking RAG Graph ---

--- 🧠: Generating Plan ---
plan:
  steps:
  - sub_question: What are the key risks related to competition as stated in NV
    tool: search_10k
    ...
  - sub_question: What are the recent news and developments in AMD's AI chip st
    tool: search_web
    ...

--- 🔍: Retrieving from 10-K (Step 1: ...) ---
  Rewritten Query: key competitive risks for NVIDIA in the semiconductor indust
  Supervisor Decision: Use `hybrid_search`. ...

--- 🎯: Reranking Documents ---
  Reranked to top 3 documents.

--- ✂️: Distilling Context ---
  Distilled Context Snippet: NVIDIA operates in the intensely competitive semic

--- 🤔: Reflecting on Findings ---
  Summary: According to its 2023 10-K, NVIDIA operates in an intensely competit

--- 🚦: Evaluating Policy ---
  -> Decision: CONTINUE_PLAN | Justification: The first step...has been complet

--- 🌐: Searching Web (Step 2: ...) ---
  Rewritten Query: AMD AI chip strategy news and developments 2024...
```

```
--- 🎯: Reranking Documents ---
  Reranked to top 3 documents.

--- ✂️: Distilling Context ---
  Distilled Context Snippet: AMD has ramped up its challenge to Nvidia in the A

--- 🤔: Reflecting on Findings ---
  Summary: In 2024, AMD is aggressively competing with NVIDIA in the AI chip ma

--- 🚦: Evaluating Policy ---
  -> Decision: FINISH | Justification: The research history now contains compre

--- ✅: Generating Final Answer with Citations ---

--- Graph Stream Finished ---
```

You can see the execution of our design. The agent:

1. **Planned:** It created the correct two-step, multi-tool plan.
2. **Executed Step 1:** It used `search_10k`, ran it through the full retrieval funnel, and reflected on the findings.
3. **Self-Critiqued:** The policy agent saw the plan was not yet complete and decided to `CONTINUE_PLAN`.
4. **Executed Step 2:** It correctly switched to the `search_web` tool, ran it through the same funnel, and reflected again.
5. **Self-Critiqued Again:** This time, the policy agent saw that all necessary information was gathered and correctly decided to `FINISH`.
6. **Synthesized:** The workflow then proceeded to the `generate_final_answer` node.

The agent has successfully navigated the complex query. Now, let's examine the final answer it produced.

### Analyzing the Final, High-Quality Answer

The agent has completed its research. The `final_state` variable now holds the complete `RAGState`, including the `final_answer`. Let's print it out and see if it successfully synthesized the information from both sources into a single, analytical response, complete with citations.

```
console.print("--- DEEP THINKING RAG FINAL ANSWER ---")
console.print(Markdown(final_state['final_answer']))
```

```
#### OUTPUT ####
--- DEEP THINKING RAG FINAL ANSWER ---
Based on an analysis of NVIDIA's 2023 10-K filing and recent news from 2024 reg

**NVIDIA's Stated Competitive Risks:**
In its 2023 10-K filing, NVIDIA identifies its operating environment as the "in

**AMD's 2024 AI Chip Strategy:**
In 2024, AMD has moved aggressively to challenge NVIDIA's dominance in the AI h

**Synthesis and Impact:**
AMD's 2024 AI chip strategy directly exacerbates the competitive risks outlined
```

This is a complete success. The answer is a deep list of analysis.

- It correctly summarizes the risks from the 10-K.
- It correctly summarizes the AMD news from the web search.
- Crucially, in the “Synthesis and Impact” section, it performs the multi-hop reasoning required by the original query, explaining *how* the latter exacerbates the former.
- Finally, it provides correct **provenance**, with citations pointing to both the internal document section and the external web URLs.

### Side by Side Comparison

Let’s put the two results side-by-side to make the difference crystal clear.

Comparison Table (Created by Fareed Khan)

This comparison provides the definitive conclusion. The architectural shift to a cyclical, tool-aware, and self-critiquing agent results in a dramatic and measurable improvement in performance on complex, real-world queries.

## Evaluation Framework and Analyzing Results

So, we have seen our advanced agent succeed anecdotally on one very hard query. But in a production environment, we need more than just a single success story. We need objective, quantitative, and automated validation.

Evaluation Framework (Created by Fareed Khan)

To achieve this, we will now build a rigorous evaluation framework using the **RAGAs** (RAG Assessment) library. We will focus on four critical metrics provided by RAGAs:

- **Context Precision & Recall:** These measure the quality of our retrieval pipeline. **Precision** asks, "Of the documents we retrieved, how many were actually relevant?" (Signal vs. Noise). **Recall** asks, "Of all the relevant documents that exist, how many did we actually find?" (Completeness).
- **Answer Faithfulness:** This measures whether the generated answer is grounded in the provided context, acting as our primary check against LLM hallucination.
- **Answer Correctness:** This is the ultimate measure of quality. It compares the generated answer to a manually crafted "ground truth" answer to assess its factual accuracy and completeness.

So, basically, to run a RAGAs evaluation, we need to prepare a dataset. This dataset will contain our challenge query, the answers generated by both our baseline and advanced pipelines, the respective contexts they used, and a "ground truth" answer that we'll write ourselves to serve as the ideal response.

```
from datasets import Dataset # From the Hugging Face datasets library, which RA
from ragas import evaluate
from ragas.metrics import (
    context_precision,
    context_recall,
    faithfulness,
    answer_correctness,
```

```
)
import pandas as pd


print("Preparing evaluation dataset...")

# This is our manually crafted, ideal answer to the complex query.
ground_truth_answer_adv = "NVIDIA's 2023 10-K lists intense competition and rap

# We need to re-run the retriever for the baseline model to get its context for
retrieved_docs_for_baseline_adv = baseline_retriever.invoke(complex_query_adv)
baseline_contexts = [[doc.page_content for doc in retrieved_docs_for_baseline_a

# For the advanced agent, we'll consolidate all the documents it retrieved acro
advanced_contexts_flat = []
for step in final_state['past_steps']:
    advanced_contexts_flat.extend([doc.page_content for doc in step['retrieved_

# We use a set to remove any duplicate documents for a cleaner evaluation.
advanced_contexts = [list(set(advanced_contexts_flat))]

# Now, we construct the dictionary that will be turned into our evaluation data
eval_data = {
    'question': [complex_query_adv, complex_query_adv], # The same question for
    'answer': [baseline_result, final_state['final_answer']], # The answers fro
    'contexts': baseline_contexts + advanced_contexts, # The contexts each syst
    'ground_truth': [ground_truth_answer_adv, ground_truth_answer_adv] # The id
}

# Create the Hugging Face Dataset object.
eval_dataset = Dataset.from_dict(eval_data)

# Define the list of metrics we want to compute.
metrics = [
    context_precision,
    context_recall,
    faithfulness,
    answer_correctness,
]
print("Running RAGAs evaluation...")

# Run the evaluation. RAGAs will call an LLM to perform the scoring for each me
result = evaluate(eval_dataset, metrics=metrics, is_async=False)
print("Evaluation complete.")

# Format the results into a clean pandas DataFrame for easy comparison.
results_df = result.to_pandas()
results_df.index = ['baseline_rag', 'deep_thinking_rag']

print("\n--- RAGAs Evaluation Results ---")
print(results_df[['context_precision', 'context_recall', 'faithfulness', 'answe
```

We are setting up a formal experiment. We gather all the necessary artifacts for our single, hard query: the question, the two different answers, the two different sets of context, and our ideal ground truth. We then feed this neatly packaged `eval_dataset` to the `ragas.evaluate` function.

Behind the scenes, RAGAs makes a series of LLM calls, asking it to act as a judge. For example, for `faithfulness`, it will ask, "Is this answer fully supported by this context?" For `answer_correctness`, it will ask ...

> How factually similar is this answer to this ground truth answer?

We can look at the numerical scores ...

```
#### OUTPUT ####
Preparing evaluation dataset...
Running RAGAs evaluation...
Evaluation complete.


--- RAGAs Evaluation Results ---
                     baseline_rag  deep_thinking_rag
context_precision        0.500000           0.890000
context_recall           0.333333           1.000000
faithfulness             1.000000           1.000000
answer_correctness       0.395112           0.991458
```

The quantitative results provide a definitive and objective verdict on the superiority of the Deep Thinking architecture.

- **Context Precision (0.50 vs 0.89):** The baseline's context was only half-relevant, as it could only retrieve general information about competition. The advanced agent's multi-step, multi-tool retrieval achieved a perfect precision score.
- **Context Recall (0.33 vs 1.00):** The baseline retriever completely missed the crucial information from the web, resulting in a very low recall score. The advanced agent's planning and tool-use ensured all necessary information was found, achieving perfect recall.

- **Faithfulness (1.00 vs 1.00):** Both systems were highly faithful. The baseline correctly stated it didn't have the information, and the advanced agent correctly used the information it found. This is a good sign for both, but faithfulness without correctness is not useful.
- **Answer Correctness (0.40 vs 0.99):** This is the ultimate measure of quality. The baseline's answer was less than 40% correct because it was missing the entire second half of the required analysis. The advanced agent's answer was nearly perfect.

### Summarizing Our Entire Pipeline

In this guide, we have gone on a complete architecture from a simple, brittle RAG pipeline to a sophisticated autonomous reasoning agent.

- We started by building a **vanilla RAG system** and demonstrated its predictable failure on a complex, multi-source query.
- We then systematically engineered a **Deep Thinking Agent**, equipping it with the ability to plan, use multiple tools, and adapt its retrieval strategy.
- We built a **multi-stage retrieval funnel** that moves from broad recall (with hybrid search) to high precision (with a cross-encoder reranker) and finally to synthesis (with a distiller agent).
- We orchestrated this entire cognitive architecture using **LangGraph**, creating a cyclical, stateful workflow that enables true multi-step reasoning.
- We implemented a **self-critique loop**, allowing the agent to recognize failure, revise its own plan, and exit gracefully when an answer cannot be found.
- Finally, we validated our success with a **production-grade evaluation**, using RAGAs to provide objective, quantitative proof of the advanced agent's superiority.

### Learned Policies with Markov Decision Processes (MDP)

Our agent have the Policy Agent that decides to `CONTINUE` or `FINISH` currently relies on an expensive, general-purpose LLM like GPT-4o for every single decision. While effective, this can be slow and costly in a production environment. The academic frontier offers a more optimized path forward.

- **RAG as a Decision Process:** We can frame our agent's reasoning loop as a **Markov Decision Process (MDP)**. In this model, each `RAGState` is a "state," and each action (`CONTINUE`, `REVISE`, `FINISH`) leads to a new state with a certain reward (e.g., finding the right answer).
- **Learning from Experience:** The thousands of successful and unsuccessful reasoning traces we log in **LangSmith** are invaluable training data. Each trace is an example of the agent navigating this MDP.
- **Training a Policy Model:** Using this data, we could apply **Reinforcement Learning** to train a much smaller, specialized **policy model**.
- **The Goal: Speed and Efficiency:** The goal would be to distill the complex reasoning of a model like GPT-4o into a compact, fine-tuned model (e.g., a 7B parameter model). This learned policy could make the `CONTINUE`/`FINISH` decision much faster and more cheaply, while being highly optimized for our specific domain. This is the core idea behind advanced research papers like DeepRAG and represents the next level of optimization for autonomous RAG systems.

Follow

## Written by Fareed Khan

61K followers · 0 following

I write on AI, https://www.linkedin.com/in/fareed-khan-dev/

## Responses (4)

Bgerby

What are your thoughts?

See all responses

## More from Fareed Khan and Level Up Coding

In Level Up Coding by Fareed Khan

## Building 17 Agentic AI Patterns and Their Role in Large-Scale AI Systems

Ensembling, Meta-Control, ToT, Reflexive, PEV and more

Sep 25 2.1K 45

In Level Up Coding by Hayk Simonyan

## How to Scale Like a Senior Engineer (Servers, DBs, LBs, SPOFs)

Most developers think scaling is complicated. They see “system design” and immediately think they need years of experience or some magical...

Oct 15 452 8

In Level Up Coding by Subodh Shetty

## Spring Boot + Decorator Pattern: A Smarter Way to Handle Cross-Cutting Concerns

Cut Boilerplate in Spring Boot: Use the Decorator Pattern for Logging, Auth, and Rate Limiting

Oct 2 107 2

In Level Up Coding by Fareed Khan

## Building a Multi-Layered Agentic Guardrail Pipeline to Reduce Hallucinations and Mitigate Risk

Layer 1 for Input, layer 2 for Planning, layer 3 for Output

Oct 6 631 7

See all from Fareed Khan

See all from Level Up Coding

## Recommended from Medium

Maninder Singh

## Building Vision Transformers (ViT) from Scratch

With enough data and compute, Vision Transformers (ViT) can out-perform CNNs on large benchmarks. I wanted to understand why—and more...

5d ago 260 3

Dr Nicolas Figay

## Knowledge Graphs and Ontologies: Beyond the Dictionary Fallacy

Most knowledge graph practitioners treat ontologies as sophisticated dictionaries—structured vocabularies and entity hierarchies...

6d ago 87 2

In Netflix TechBlog by Netflix Technology Blog

## How and Why Netflix Built a Real-Time Distributed Graph: Part 1—Ingesting and Processing Data...

Authors: Adrian Taruc and James Dalton

6d ago 647 16

In Level Up Coding by Fareed Khan

## Building 17 Agentic AI Patterns and Their Role in Large-Scale AI Systems

Ensembling, Meta-Control, ToT, Reflexive, PEV and more

Sep 25 2.1K 45

In CodeToDeploy by TechToFit - Master Your Life with Tech

## I Tried Google's New AI Agents. It's a Gold Rush.

I spend my days deep in the world of AI, but every so often, something drops that makes me stop everything. This is one of those times...

Oct 10 164 3

In Artificial Intelligence in Plain English by Alpha Iterations

## Build Agentic RAG using LangGraph

A practical guide to build Agentic RAG with complete project code

5d ago 14

See more recommendations