Building RAG Research Multi-Agent with LangGraph

Nicola Disabato · Follow
22 min read · Jan 11, 2025

Listen Share More

Image credits: https://infohub.delltechnologies.com/it-it/p/the-rise-of-agentic-rag-systems/

In this article, we present a practical project featuring a **RAG Research Multi-Agent** tool developed using **LangGraph**. This tool is designed to address **complex questions** requiring **multiple sources** and **iterative steps** to arrive at a final answer. It uses an **hybrid search** and a **Cohere reraking step** for retrieving documents, and also incorporates a **self-corrective mechanism**, including a **hallucination-check** process, to improve responses reliability, making it ideal for enterprise applications.

Github Repo here.

1. Introduction — Naive vs Agentic RAG

For the purpose of the project, a Naive RAG approach is not sufficient for the following reasons:

- **No Complex Query Understanding**: Inability to break down a complex query into multiple manageable sub-steps, processing the query at a single level instead of analyzing each step and arriving at a unified conclusion.
- **Lack of hallucinations or error handling**: Naive RAG pipelines lack a response verification step and mechanisms to handle hallucinations, preventing them from correcting errors by generating a new response.
- **Lack of dynamic tool use**: A Naive RAG system does not allow the use of tools, calling external APIs, or interacting with databases based on workflow conditions.

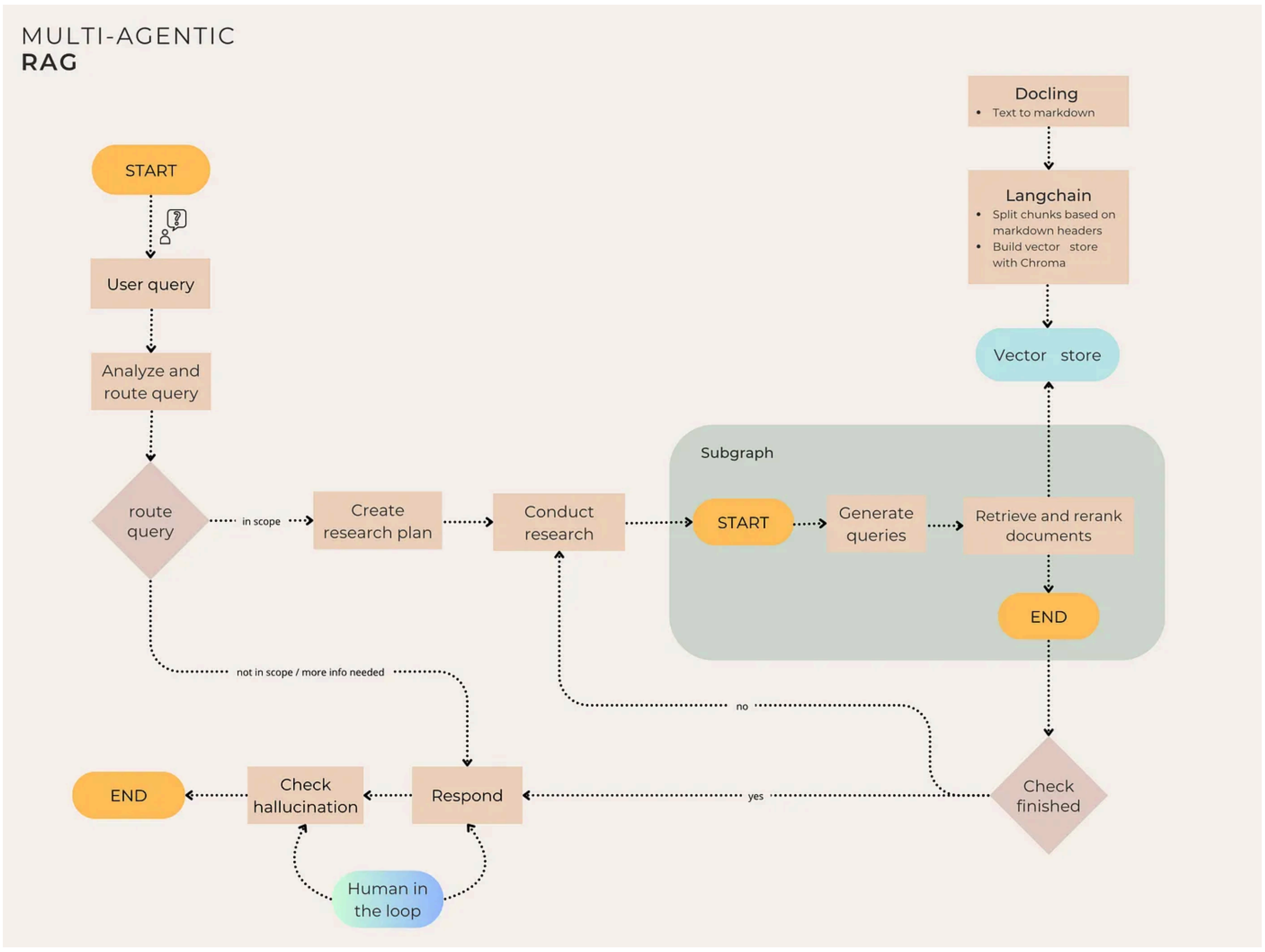
An **Multi-Agent RAG research system** was therefore implemented to address all these issues. An agent-based framework, in fact, allows for:

- **Routing and using tools**: A **Routing agent** can classify the user's query and direct the flow to the appropriate node or tool. This enables context-driven decisions, such as determining whether a document requires a full summarization, if more detailed informations are required, or if the question is out of scope.
- **Planning sub-steps**: Complex queries often need to be broken down into smaller, manageable steps. Starting from a query, it is possible to generate a list of steps to execute in order to reach a conclusion while exploring different facets of the query. For instance, if a query requires a comparison between two different sections of a document, an agent-based approach would allow recognizing this need for comparison, retrieving both sources separately, and merging them into a comparative analysis in the final response.
- **Reflection and Error Correction**: In addition to the simple response generation, an agent-based approach would allow for adding a validation step to address potential hallucinations, errors, or responses that fail to accurately answer the user's query. This also enables the integration of a **self-correction** mechanism with **human-in-the-loop**, which incorporates human input into automated processes. Such functionality makes agent-based RAG systems a more robust

and reliable solution for enterprise applications, where reliability is a top priority.

- **Shared Global State:** An Agent Workflow shares a global state, simplifying the management of states across multiple steps. This shared state is essential for maintaining consistency across different stages of a multi-agent process.

2. Project overview



Agentic RAG Diagram

Graph steps:

1. **Analyze and route query (Adaptive RAG):** The user's query is classified and routed to the appropriate node. From there, the system can either **proceed to the next step** (the "research plan generation"), **request more information** from the user, or **respond immediately** if the query is **out of scope**.

2. **Research Plan generation:** The system generates a step-by-step research plan, one or more steps depending on the complexity of the request. It then returns a

list of specific steps required to address the user's question.

3. **Research subgraph:** For each step defined in the research plan generation a subgraph is called. Specifically, the subgraph starts generating two queries via LLM. Next, the system retrieves documents relevant to these generated queries by using an **ensemble retriever** (using **similarity search**, **BM25**, and **MMR**). A **Reranking Step** then applies **Cohere-based contextual compression**, ultimately yielding the top *k* pertinent documents for all steps with their relevant scores.
4. **Generation Step:** Basing on the relevant documents the tool generates an answer via an LLM.
5. **Hallucination Check (Self-Corrective RAG with Human-in-the-Loop): There is a reflection step where t**he system analyzes the generated answer to determine whether it is supported by the provided context and addresses all aspects. If the check fails, the graph workflow is interrupted and the user is prompted to either generate a revised answer or end the process.

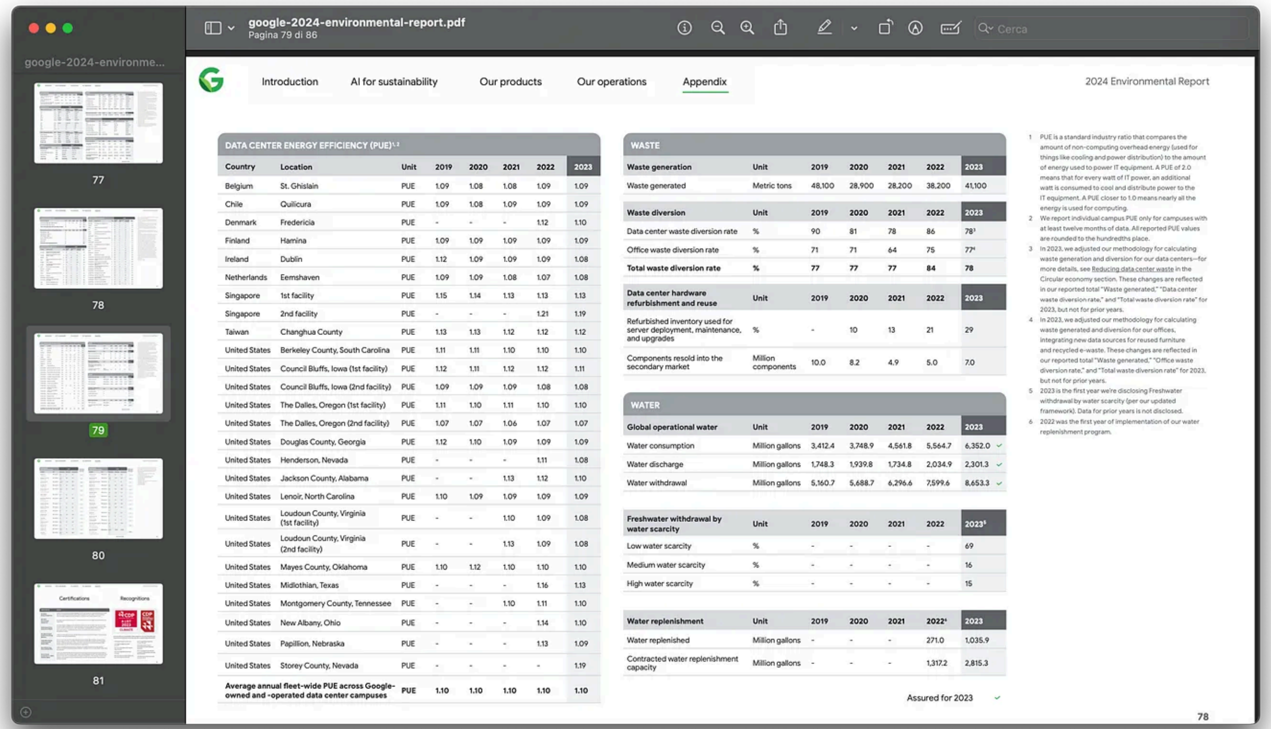
For the creation of the vector store, a paragraph-based chunking approach is implemented using **Docling** and **LangChain**, and the vector database is built with **ChromaDB**.

Building vector DB

Document parsing

For PDFs with complex structures, including tables with complex layouts, it is essential to carefully select the tool used for parsing. Many libraries lack precision when processing PDFs with intricate page layouts or tabular structures.

To address this, **Docling**, an open-source library, was utilized. It enables straightforward and efficient document parsing, allowing export to the desired format. It can read and export to Markdown and JSON from a variety of commonly used document formats, including PDF, DOCX, PPTX, XLSX, images, HTML, AsciiDoc, and Markdown. Docling offers comprehensive understanding of PDF documents, including table structures, reading sequences, and page layouts. Additionally, it supports OCR for scanned PDFs.


google-2024-environmental-report.pdf
Pagina 79 di 86

Introduction | AI for sustainability | Our products | Our operations | Appendix

2024 Environmental Report

DATA CENTER ENERGY EFFICIENCY (PUE)[1, 2]

Country	Location	Unit	2019	2020	2021	2022	2023
Belgium	St. Ghislain	PUE	1.09	1.08	1.08	1.09	1.09
Chile	Quilicura	PUE	1.09	1.08	1.09	1.09	1.09
Denmark	Fredericia	PUE	-	-	-	1.12	1.10
Finland	Hamina	PUE	1.09	1.09	1.09	1.09	1.09
Ireland	Dublin	PUE	1.12	1.09	1.09	1.09	1.08
Netherlands	Eemshaven	PUE	1.09	1.09	1.08	1.07	1.08
Singapore	1st facility	PUE	1.15	1.14	1.13	1.13	1.13
Singapore	2nd facility	PUE	-	-	-	1.21	1.19
Taiwan	Changhua County	PUE	1.13	1.13	1.12	1.12	1.12
United States	Berkeley County, South Carolina	PUE	1.11	1.11	1.10	1.10	1.10
United States	Council Bluffs, Iowa (1st facility)	PUE	1.12	1.11	1.12	1.12	1.11
United States	Council Bluffs, Iowa (2nd facility)	PUE	1.09	1.09	1.09	1.08	1.08
United States	The Dalles, Oregon (1st facility)	PUE	1.11	1.10	1.11	1.10	1.10
United States	The Dalles, Oregon (2nd facility)	PUE	1.07	1.07	1.06	1.07	1.07
United States	Douglas County, Georgia	PUE	1.12	1.10	1.09	1.09	1.09
United States	Henderson, Nevada	PUE	-	-	-	1.11	1.08
United States	Jackson County, Alabama	PUE	-	-	1.13	1.12	1.10
United States	Lenoir, North Carolina	PUE	1.10	1.09	1.09	1.09	1.09
United States	Loudoun County, Virginia (1st facility)	PUE	-	-	1.10	1.09	1.08
United States	Loudoun County, Virginia (2nd facility)	PUE	-	-	1.13	1.09	1.08
United States	Mayes County, Oklahoma	PUE	1.10	1.12	1.10	1.10	1.10
United States	Midlothian, Texas	PUE	-	-	-	1.16	1.13
United States	Montgomery County, Tennessee	PUE	-	-	1.10	1.11	1.10
United States	New Albany, Ohio	PUE	-	-	-	1.14	1.10
United States	Papillion, Nebraska	PUE	-	-	-	1.13	1.09
United States	Storey County, Nevada	PUE	-	-	-	-	1.19
Average annual fleet-wide PUE across Google-owned and -operated data center campuses		**PUE**	**1.10**	**1.10**	**1.10**	**1.10**	**1.10**

WASTE

Waste generation	Unit	2019	2020	2021	2022	2023
Waste generated	Metric tons	48,100	28,900	28,200	38,200	41,100

Waste diversion	Unit	2019	2020	2021	2022	2023
Data center waste diversion rate	%	90	81	78	86	78[3]
Office waste diversion rate	%	71	71	64	75	77[4]
Total waste diversion rate	**%**	**77**	**77**	**77**	**84**	**78**

Data center hardware refurbishment and reuse	Unit	2019	2020	2021	2022	2023
Refurbished inventory used for server deployment, maintenance, and upgrades	%	-	10	13	21	29
Components resold into the secondary market	Million components	10.0	8.2	4.9	5.0	7.0

WATER

Global operational water	Unit	2019	2020	2021	2022	2023
Water consumption	Million gallons	3,412.4	3,748.9	4,561.8	5,564.7	6,352.0 ✓
Water discharge	Million gallons	1,748.3	1,939.8	1,734.8	2,034.9	2,301.3 ✓
Water withdrawal	Million gallons	5,160.7	5,688.7	6,296.6	7,599.6	8,653.3 ✓

Freshwater withdrawal by water scarcity	Unit	2019	2020	2021	2022	2023[5]
Low water scarcity	%	-	-	-	-	69
Medium water scarcity	%	-	-	-	-	16
High water scarcity	%	-	-	-	-	15

Water replenishment	Unit	2019	2020	2021	2022[6]	2023
Water replenished	Million gallons	-	-	-	271.0	1,035.9
Contracted water replenishment capacity	Million gallons	-	-	-	1,317.2	2,815.3

Assured for 2023 ✓

1 PUE is a standard industry ratio that compares the amount of non-computing overhead energy (used for things like cooling and power distribution) to the amount of energy used to power IT equipment. A PUE of 2.0 means that for every watt of IT power, an additional watt is consumed to cool and distribute power to the IT equipment. A PUE closer to 1.0 means nearly all the energy is used for computing.

2 We report individual campus PUE only for campuses with at least twelve months of data. All reported PUE values are rounded to the hundredths place.

3 In 2023, we adjusted our methodology for calculating waste generation and diversion for our data centers—for more details, see Reducing data center waste in the Circular economy section. These changes are reflected in our reported total "Waste generated," "Data center waste diversion rate," and "Total waste diversion rate" for 2023, but not for prior years.

4 In 2023, we adjusted our methodology for calculating waste generated and diversion for our offices, integrating new data sources for reused furniture and recycled e-waste. These changes are reflected in our reported total "Waste generated," "Office waste diversion rate," and "Total waste diversion rate" for 2023, but not for prior years.

5 2023 is the first year we're disclosing Freshwater withdrawal by water scarcity (per our updated framework). Data for prior years is not disclosed.

6 2022 was the first year of implementation of our water replenishment program.

78

Pdf preview

The text contained in the PDF was then converted to Markdown format, which is necessary for chunking that follows a paragraph-based structure.

```
from docling.document_converter import DocumentConverter

logger.info("Starting document processing.")
converter = DocumentConverter()
markdown_document = converter.convert(source).document.export_to_markdown()
```

The extracted text will have a structure similar to the following image. As can be seen, the PDF and table parsing has extracted text that retains the original formatting.

```
## WATER

| Global operational water   | Unit           | 2019    | 2020    | 2021    | 2022    | 2023    |
|----------------------------|----------------|---------|---------|---------|---------|---------|
| Water consumption          | Million gallons | 3,412.4 | 3,748.9 | 4,561.8 | 5,564.7 | 6,352.0 |
| Water discharge            | Million gallons | 1,748.3 | 1,939.8 | 1,734.8 | 2,034.9 | 2,301.3 |
| Water withdrawal           | Million gallons | 5,160.7 | 5,688.7 | 6,296.6 | 7,599.6 | 8,653.3 |

| Freshwater withdrawal by   water scarcity   | Unit   | 2019   | 2020   | 2021   | 2022   |   2023 5 |
|---------------------------------------------|--------|--------|--------|--------|--------|----------|
| Low water scarcity                          | %      | -      | -      | -      | -      |       69 |
| Medium water scarcity                       | %      | -      | -      | -      | -      |       16 |
| High water scarcity                         | %      | -      | -      | -      | -      |       15 |

| Water replenishment                     | Unit            | 2019   | 2020   | 2021   | 2022 6    | 2023    |
|-----------------------------------------|-----------------|--------|--------|--------|-----------|---------|
| Water replenished                       | Million gallons | -      | -      | -      | 271.0     | 1,035.9 |
| Contracted water replenishment  capacity | Million gallons | -      | -      | -      | 1,317.2   | 2,815.3 |

- 1 PUE is a standard industry ratio that compares the amount of non-computing overhead energy (used for things like cool
- 2 We report individual campus PUE only for campuses with at least twelve months of data. All reported PUE values are ro
- 3 In 2023, we adjusted our methodology for calculating waste generation and diversion for our data centers-for more det
- 4 In 2023, we adjusted our methodology for calculating waste generated and diversion for our offices, integrating new d
- 5 2023 is the first year we're disclosing Freshwater withdrawal by water scarcity (per our updated framework). Data for
- 6 2022 was the first year of implementation of our water replenishment program.

<!-- image -->
```

Based on the headers and using `MarkdownHeaderTextSplitter` , the output text was subsequently split into chunks, resulting in a list of 332 `Document` objects (*LangChain Document*).

```python
from langchain_text_splitters import MarkdownHeaderTextSplitter

headers_to_split_on = [
    ("#", "Header 1"),
    ("##", "Header 2")
]

markdown_splitter = MarkdownHeaderTextSplitter(headers_to_split_on)
docs_list = markdown_splitter.split_text(markdown_document)
docs_list
```

```
# Output example
[Document(metadata={'Header 2': 'A letter from our Chief Sustainability Officer
...]

# len(docs_list):
332
```

Vector store building

We build a vector database to store sentences as vector embeddings and search through that database. In this case, we use **Chroma** and store a persistent db in the local directory ‘`db_vector`’.

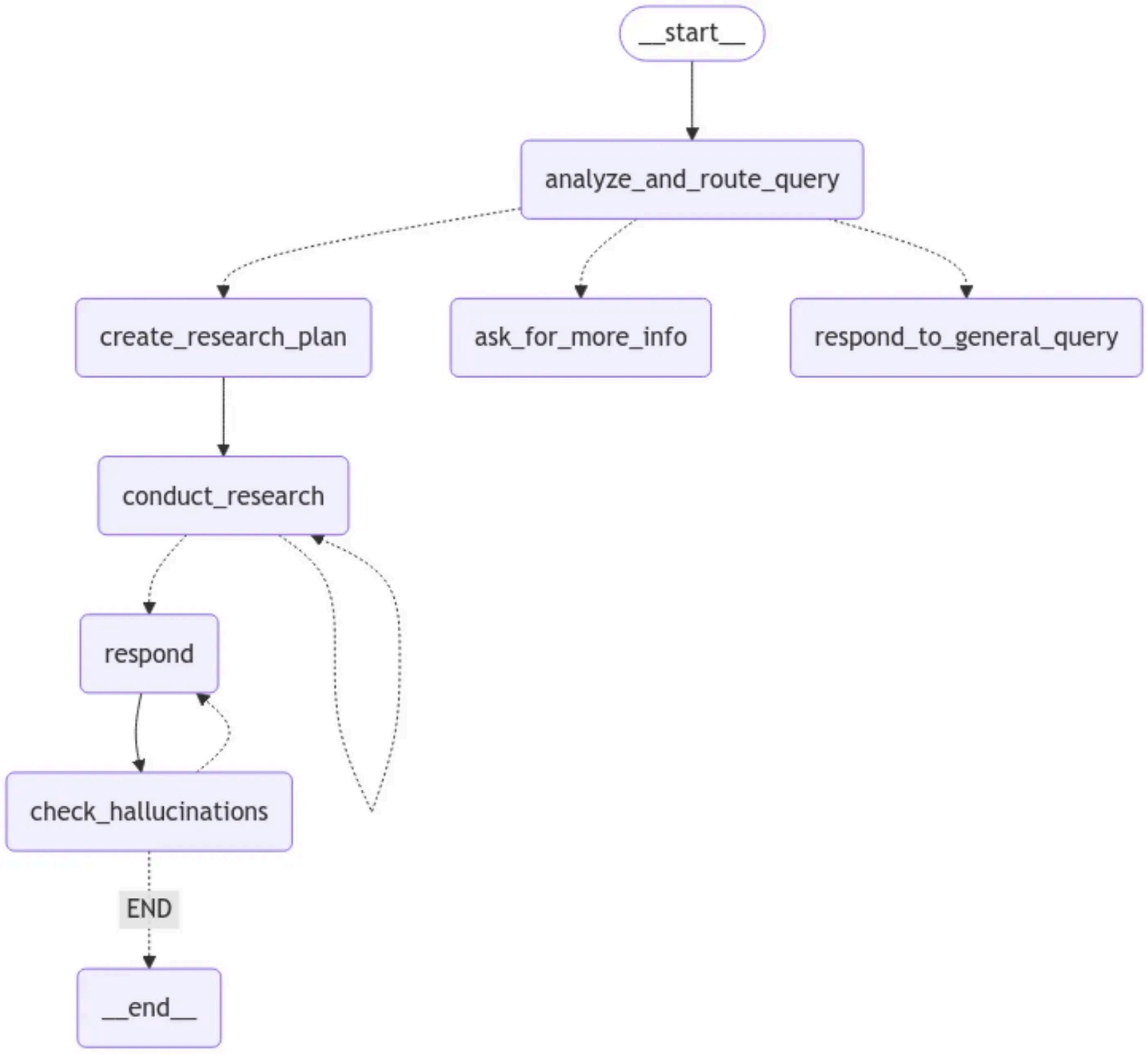
```
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings

embd = OpenAIEmbeddings()

vectorstore_from_documents = Chroma.from_documents(
    documents=docs_list,
    collection_name="rag-chroma-google-v1",
    embedding=embd,
    persist_directory='db_vector'
)
```

Main Graph building

The implemented system includes two graphs:

- A **researcher graph** as a **subgraph**, tasked with generating different queries that will be used for retrieving and re-ranking the top-k documents from the vector database.
- A **main graph**, which contains the primary workflow, such as analyzing the user’s query, generating the steps required to complete a task, producing the response, and checking for hallucinations with a human-in-the-loop mechanism.

Main Graph structure



LangGraph graph preview

One of the central concepts of **LangGraph is state**. Each graph execution creates a state that is passed between nodes in the graph as they execute, and each node updates this internal state with its return value after it executes.

Let's start the project with building the graph states. To achieve this, we define the two classes:

- **Router:** Contains the result of classifying the user's query into one of the classes: "more-info," "environmental," or "general."
- **GradeHallucination:** Contains a binary score indicating the presence of hallucinations in the response.

```
from pydantic import BaseModel, Field

class Router(TypedDict):
    """Classify user query."""

    logic: str
    type: Literal["more-info", "environmental", "general"]

from pydantic import BaseModel, Field

class GradeHallucinations(BaseModel):
    """Binary score for hallucination present in generation answer."""

    binary_score: str = Field(
        description="Answer is grounded in the facts, '1' or '0'"
    )
```

The defined graph states are:

- **InputState:** Includes the list of messages exchanged between the user and the agent.
- **AgentState:** Contains the `Router`'s classification of the user's query, the list of steps to execute in the Research Plan, the list of retrieved documents that the agent can reference, and the binary score `Gradehallucination`.

```
from dataclasses import dataclass, field
from typing import Annotated, Literal, TypedDict
from langchain_core.documents import Document
from langchain_core.messages import AnyMessage
from langgraph.graph import add_messages
from utils.utils import reduce_docs

@dataclass(kw_only=True)
class InputState:
    """Represents the input state for the agent.

    This class defines the structure of the input state, which includes
    the messages exchanged between the user and the agent. It serves as
    a restricted version of the full State, providing a narrower interface
    to the outside world compared to what is maintained iternally.
    """

    messages: Annotated[list[AnyMessage], add_messages]
    """Messages track the primary execution state of the agent.

    Typically accumulates a pattern of Human/AI/Human/AI messages.

    Returns:
        A new list of messages with the messages from `right` merged into `left
        If a message in `right` has the same ID as a message in `left`, the
        message from `right` will replace the message from `left`."""



# Primary agent state
@dataclass(kw_only=True)
class AgentState(InputState):
    """State of the retrieval graph / agent."""

    router: Router = field(default_factory=lambda: Router(type="general", logic
    """The router's classification of the user's query."""
    steps: list[str] = field(default_factory=list)
    """A list of steps in the research plan."""
    documents: Annotated[list[Document], reduce_docs] = field(default_factory=l
    """Populated by the retriever. This is a list of documents that the agent c
    hallucination: GradeHallucinations = field(default_factory=lambda: GradeHal
```

Step 1: Analyze and route query

The function `analyze_and_route_query` returns and updates the `router` variable of the state `AgentState`. The function `route_query` determines the next step based on the previous query classification

Specifically, this step updates the state with a `Router` object whose `type` variable contains one of the following values: `"more-info"`, `"environmental"`, or `"general"`. Based on this information, the workflow will be routed to the appropriate node (one of `"create_research_plan"`, `"ask_for_more_info"`, or `"respond_to_general_query"`).

```
async def analyze_and_route_query(
    state: AgentState, *, config: RunnableConfig
) -> dict[str, Router]:
    """Analyze the user's query and determine the appropriate routing.

    This function uses a language model to classify the user's query and decide
    within the conversation flow.

    Args:
        state (AgentState): The current state of the agent, including conversat
        config (RunnableConfig): Configuration with the model used for query ar

    Returns:
        dict[str, Router]: A dictionary containing the 'router' key with the cl
    """
    model = ChatOpenAI(model=GPT_4o, temperature=TEMPERATURE, streaming=True)
    messages = [
        {"role": "system", "content": ROUTER_SYSTEM_PROMPT}
    ] + state.messages
    logging.info("---ANALYZE AND ROUTE QUERY---")
    response = cast(
        Router, await model.with_structured_output(Router).ainvoke(messages)
    )
    return {"router": response}


def route_query(
    state: AgentState,
) -> Literal["create_research_plan", "ask_for_more_info", "respond_to_general_c
    """Determine the next step based on the query classification.

    Args:
        state (AgentState): The current state of the agent, including the route

    Returns:
        Literal["create_research_plan", "ask_for_more_info", "respond_to_genera

    Raises:
        ValueError: If an unknown router type is encountered.
    """
    _type = state.router["type"]
    if _type == "environmental":
        return "create_research_plan"
    elif _type == "more-info":
        return "ask_for_more_info"
    elif _type == "general":
        return "respond_to_general_query"
    else:
        raise ValueError(f"Unknown router type {_type}")
```

Output example to the question *"Retrieve the data center PUE efficiency value in Dublin in 2019"*:

```
{
  "logic":"This is a specific question about the environmental efficiency of a
```

```
    "type":"environmental"
  }
```

Step 1.1 Out of scope / More informations needed

We then define the functions `ask_for_more_info` and `respond_to_general_query`, which directly generate a response for the user by making a call to the LLM: the first will be executed if the router determines that more information is needed from the user, while the second generates a response to a general query not related to our topic. In this case, it is necessary to concatenate the generated response to the list of messages, updating the `messages` variable in the state.

```
async def ask_for_more_info(
    state: AgentState, *, config: RunnableConfig
) -> dict[str, list[BaseMessage]]:
    """Generate a response asking the user for more information.

    This node is called when the router determines that more information is nee

    Args:
        state (AgentState): The current state of the agent, including conversat
        config (RunnableConfig): Configuration with the model used to respond.

    Returns:
        dict[str, list[str]]: A dictionary with a 'messages' key containing the
    """
    model = ChatOpenAI(model=GPT_4o_MINI, temperature=TEMPERATURE, streaming=Tr
    system_prompt = MORE_INFO_SYSTEM_PROMPT.format(
        logic=state.router["logic"]
    )
    messages = [{"role": "system", "content": system_prompt}] + state.messages
    response = await model.ainvoke(messages)
    return {"messages": [response]}



async def respond_to_general_query(
    state: AgentState, *, config: RunnableConfig
) -> dict[str, list[BaseMessage]]:
    """Generate a response to a general query not related to environmental.

    This node is called when the router classifies the query as a general quest

    Args:
        state (AgentState): The current state of the agent, including conversat
        config (RunnableConfig): Configuration with the model used to respond.
    Returns:
        dict[str, list[str]]: A dictionary with a 'messages' key containing the
    """
    model = ChatOpenAI(model=GPT_4o_MINI, temperature=TEMPERATURE, streaming=Tr
    system_prompt = GENERAL_SYSTEM_PROMPT.format(
        logic=state.router["logic"]
    )
    logging.info("---RESPONSE GENERATION---")
    messages = [{"role": "system", "content": system_prompt}] + state.messages
    response = await model.ainvoke(messages)
    return {"messages": [response]}
```

Output example to the question *"What's the weather like in Altamura?":*

```
{
  "logic":"What's the weather like in Altamura?",
  "type":"general"
}
```

```
# ---RESPONSE GENERATION---
"I appreciate your question, but I'm unable to provide information about the we
```

Step 2: Create a research plan

If the query classification returns the value `"environmental"`, the user's request is in scope with the document, and the workflow will reach the `create_research_plan` node, whose function creates a step-by-step research plan for answering an environmental-related query.

```
async def create_research_plan(
    state: AgentState, *, config: RunnableConfig
) -> dict[str, list[str] | str]:
    """Create a step-by-step research plan for answering a environmental-relate

    Args:
        state (AgentState): The current state of the agent, including conversat
        config (RunnableConfig): Configuration with the model used to generate
    Returns:
        dict[str, list[str]]: A dictionary with a 'steps' key containing the li
    """

    class Plan(TypedDict):
        """Generate research plan."""

        steps: list[str]

    model = ChatOpenAI(model=GPT_4o_MINI, temperature=TEMPERATURE, streaming=Tr
    messages = [
        {"role": "system", "content": RESEARCH_PLAN_SYSTEM_PROMPT}
    ] + state.messages
    logging.info("---PLAN GENERATION---")
    response = cast(Plan, await model.with_structured_output(Plan).ainvoke(mess
    return {"steps": response["steps"], "documents": "delete"}
```

Output example to the question *"Retrieve the data center PUE efficiency value in Dublin in 2019":*

```
{
  "steps":
    ["Look up the PUE (Power Usage Effectiveness) efficiency value for data cer
  ]
}
```

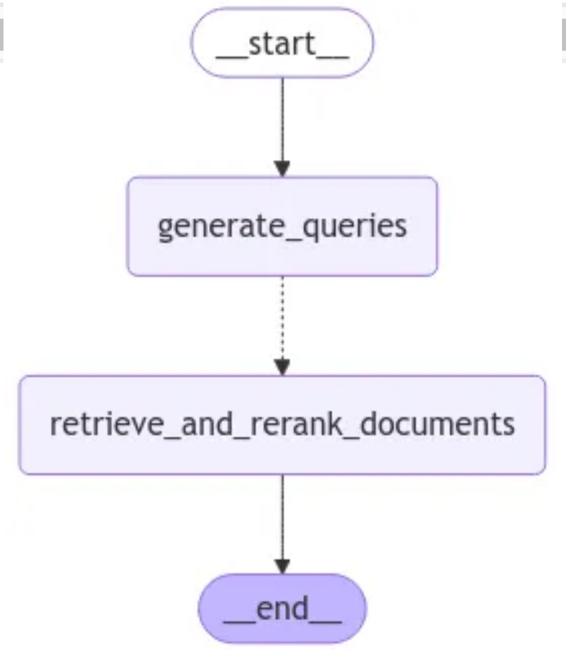
In this case, the user's request only requires one step to retrieve the information.

Step 3: Conduct research

This function takes the first step from the research plan and uses it to conduct research. For the research, the function calls the **subgraph** `researcher_graph`, which returns a list of chunks that we will explore in the next section. Finally, we update the `steps` variable in the state by removing the step that was just executed.

```
async def conduct_research(state: AgentState) -> dict[str, Any]:
    """Execute the first step of the research plan.

    This function takes the first step from the research plan and uses it to cc
    Args:
        state (AgentState): The current state of the agent, including the resea

    Returns:
        dict[str, list[str]]: A dictionary with 'documents' containing the rese
                              'steps' containing the remaining research steps.

    Behavior:
        - Invokes the researcher_graph with the first step of the research plar
        - Updates the state with the retrieved documents and removes the comple
    """
    result = await researcher_graph.ainvoke({"question": state.steps[0]}) #grap
    docs = result["documents"]
    step = state.steps[0]
    logging.info(f"\n{len(docs)} documents retrieved in total for the step: {st
    return {"documents": result["documents"], "steps": state.steps[1:]}
```

Step 4: Researcher subgraph building



As visible in the image above, the graph consists of a query generation step, starting from the step passed by the main graph, and a retrieving step for the relevant chunks. As we did for the main graph, let’s proceed with defining the states `QueryState` (private state for the `retrieve_documents` node in the researcher graph) and `ResearcherState` (state of the researcher graph).

```
"""States for the researcher subgraph.

This module defines the state structures used in the researcher subgraph.
"""

from dataclasses import dataclass, field
from typing import Annotated
from langchain_core.documents import Document
from utils.utils import reduce_docs

@dataclass(kw_only=True)
class QueryState:
    """Private state for the retrieve_documents node in the researcher graph.""
    query: str

@dataclass(kw_only=True)
class ResearcherState:
    """State of the researcher graph / agent."""
    question: str
    """A step in the research plan generated by the retriever agent."""
    queries: list[str] = field(default_factory=list)
    """A list of search queries based on the question that the researcher gener
    documents: Annotated[list[Document], reduce_docs] = field(default_factory=l
    """Populated by the retriever. This is a list of documents that the agent c
```

Step 4.1: Generate queries

This step generate search queries based on the question (a step in the research plan). This function uses a LLM to generate diverse search queries to help answer the question.

```
async def generate_queries(
    state: ResearcherState, *, config: RunnableConfig
) -> dict[str, list[str]]:
    """Generate search queries based on the question (a step in the research pl

    This function uses a language model to generate diverse search queries to h

    Args:
        state (ResearcherState): The current state of the researcher, including
        config (RunnableConfig): Configuration with the model used to generate

    Returns:
        dict[str, list[str]]: A dictionary with a 'queries' key containing the
    """
    class Response(TypedDict):
        queries: list[str]

    logger.info("---GENERATE QUERIES---")
    model = ChatOpenAI(model="gpt-4o-mini-2024-07-18", temperature=0)
    messages = [
        {"role": "system", "content": GENERATE_QUERIES_SYSTEM_PROMPT},
        {"role": "human", "content": state.question},
    ]
    response = cast(Response, await model.with_structured_output(Response).ainv
    queries = response["queries"]
    queries.append(state.question)
    logger.info(f"Queries: {queries}")
    return {"queries": response["queries"]}
```

Output example to the question *"Retrieve the data center PUE efficiency value in Dublin in 2019":*

```
{
  "queries":[
    "Look up the PUE (Power Usage Effectiveness) efficiency value for data cent
    "PUE efficiency value data centers Dublin 2019",
    "Power Usage Effectiveness statistics data centers Dublin 2019"
  ]
}
```

Once the queries are generated, we can define the vectorstore using the persistent database defined earlier.

```
def _setup_vectorstore() -> Chroma:
    """
    Set up and return the Chroma vector store instance.
    """
    embeddings = OpenAIEmbeddings()
    return Chroma(
        collection_name=VECTORSTORE_COLLECTION,
        embedding_function=embeddings,
        persist_directory=VECTORSTORE_DIRECTORY
    )
```

In a RAG systems, **the most crucial part is the document retrieval process.** For this reason, significant attention has been given to the techniques used: specifically, an **ensemble retriever** was chosen as Hybrid Search and **Cohere** for reranking.

Hybrid Search is a combination of “keyword style” search and a “vector style” search. It has the advantage of doing keyword search as well as the advantage of doing a semantic search that we get from embeddings and a vector search. Ensemble Retriever is a retrieval algorithm designed to enhance the performance of information retrieval by combining the strengths of multiple individual retrievers. This approach, known as “ensemble retrieval,” uses a method called Reciprocal Rank Fusion to rerank and merge the results from different retrievers, thereby providing more accurate and relevant results than any single retriever alone.

```
# Create base retrievers
retriever_bm25 = BM25Retriever.from_documents(documents, search_kwargs={"k": TO
retriever_vanilla = vectorstore.as_retriever(search_type="similarity", search_k
retriever_mmr = vectorstore.as_retriever(search_type="mmr", search_kwargs={"k":

ensemble_retriever = EnsembleRetriever(
        retrievers=[retriever_vanilla, retriever_mmr, retriever_bm25],
        weights=ENSEMBLE_WEIGHTS,
    )
```

Reranking is a technique that can be used to improve the performance of RAG pipelines. It is a very powerful method which can significantly boost search systems. In short, reranking takes a query and a response, and outputs a **relevance score** between them. In that way, one can use any search system to surface a number of documents that can potentially contain the answer to a query, and then sort them using the Rerank endpoint.

But: why do we need a Reranking step?

To address the challenges with accuracy, two-stage retrieval was used as a means of increasing search quality. In these two-stage systems, a first-stage model (ensemble retriever) retrieves a set of candidate documents from a larger dataset. Then, a second-stage model (the reranker) is used to rerank those documents retrieved by the first-stage model. Moreover, a Reranking model, such as Cohere Rerank, is a type of model that will output a similarity score when given a query and document pair. This score can be used to reorder the documents that are most relevant to the search query. Among the reranking methodologies, the Cohere Rerank model stands out for its ability to significantly enhance search accuracy. The model diverges from traditional embedding models by employing deep learning to evaluate the alignment between each document and the query directly. Cohere Rerank outputs a relevance score by processing the query and document in tandem, which results in a more nuanced document selection process. *(full reference here)*

In this case, the retrieved documents are reranked, and the top 2 most relevant are returned.

```
from langchain.retrievers.contextual_compression import ContextualCompressionRe
from langchain_cohere import CohereRerank
from langchain_community.llms import Cohere

# Set up Cohere re-ranking
compressor = CohereRerank(top_n=2, model="rerank-english-v3.0")

# Build compression retriever
compression_retriever = ContextualCompressionRetriever(
    base_compressor=compressor,
    base_retriever=ensemble_retriever,
)

compression_retriever.invoke(
    "Retrieve the data center PUE efficiency in Dublin in 2019"
)
```

Output example to the question *"Retrieve the data center PUE efficiency value in Dublin in 2019"*:

```
[Document(metadata={'Header 2': 'Endnotes', 'relevance_score': 0.27009502}, pag
 Document(metadata={'Header 2': 'DATA CENTER GRID REGION CFE', 'relevance_score
```

Step 4.2: Retrieve and rerank documents function

```
async def retrieve_and_rerank_documents(
    state: QueryState, *, config: RunnableConfig
) -> dict[str, list[Document]]:
    """Retrieve documents based on a given query.

    This function uses a retriever to fetch relevant documents for a given quer

    Args:
        state (QueryState): The current state containing the query string.
        config (RunnableConfig): Configuration with the retriever used to fetch

    Returns:
        dict[str, list[Document]]: A dictionary with a 'documents' key containi
    """
    logger.info("---RETRIEVING DOCUMENTS---")
    logger.info(f"Query for the retrieval process: {state.query}")

    response = compression_retriever.invoke(state.query)

    return {"documents": response}
```

Step 4.3 Building subgraph

```
builder = StateGraph(ResearcherState)
builder.add_node(generate_queries)
builder.add_node(retrieve_and_rerank_documents)
builder.add_edge(START, "generate_queries")
builder.add_conditional_edges(
    "generate_queries",
    retrieve_in_parallel,  # type: ignore
    path_map=["retrieve_and_rerank_documents"],
)
builder.add_edge("retrieve_and_rerank_documents", END)
researcher_graph = builder.compile()
```

Step 5: Check finished

Using a `conditional_edge`, we build a loop with the end condition determined by the value returned by `check_finished`. This function checks that there are no more steps to process in the list of steps created by the `create_research_plan` node. Once all steps are completed, the flow proceeds to the `respond` node.

```
def check_finished(state: AgentState) -> Literal["respond", "conduct_research"]
    """Determine if the research process is complete or if more research is nee

    This function checks if there are any remaining steps in the research plan:
        - If there are, route back to the `conduct_research` node
        - Otherwise, route to the `respond` node

    Args:
        state (AgentState): The current state of the agent, including the remai

    Returns:
        Literal["respond", "conduct_research"]: The next step to take based on
    """
    if len(state.steps or []) > 0:
        return "conduct_research"
    else:
        return "respond"
```

Step 6: Respond

Generates a final response to the user's query based on the conducted research. This function formulates a comprehensive answer using the conversation history and the documents retrieved by the researcher agent.

```
async def respond(
    state: AgentState, *, config: RunnableConfig
) -> dict[str, list[BaseMessage]]:
    """Generate a final response to the user's query based on the conducted res

    This function formulates a comprehensive answer using the conversation hist

    Args:
        state (AgentState): The current state of the agent, including retrievec
        config (RunnableConfig): Configuration with the model used to respond.

    Returns:
        dict[str, list[str]]: A dictionary with a 'messages' key containing the
    """
    print("--- RESPONSE GENERATION STEP ---")
    model = ChatOpenAI(model="gpt-4o-2024-08-06", temperature=0)
    context = format_docs(state.documents)
    prompt = RESPONSE_SYSTEM_PROMPT.format(context=context)
    messages = [{"role": "system", "content": prompt}] + state.messages
    response = await model.ainvoke(messages)

    return {"messages": [response]}
```

Step 7: Check hallucination

This step checks if the response generated by LLM in the previous step is supported by the set of facts based on the document retrieved, giving a binary score.

```
async def check_hallucinations(
    state: AgentState, *, config: RunnableConfig
) -> dict[str, Any]:
    """Analyze the user's query and checks if the response is supported by the
    providing a binary score result.

    This function uses a language model to analyze the user's query and gives a

    Args:
        state (AgentState): The current state of the agent, including conversat
        config (RunnableConfig): Configuration with the model used for query ar

    Returns:
        dict[str, Router]: A dictionary containing the 'router' key with the cl
    """

    model = ChatOpenAI(model=GPT_4o_MINI, temperature=TEMPERATURE, streaming=Tr
    system_prompt = CHECK_HALLUCINATIONS.format(
        documents=state.documents,
        generation=state.messages[-1]
    )

    messages = [
        {"role": "system", "content": system_prompt}
    ] + state.messages
    logging.info("---CHECK HALLUCINATIONS---")
    response = cast(GradeHallucinations, await model.with_structured_output(Gra

    return {"hallucination": response}
```

Step 8: Human approval (human-in-the-loop)

If the LLM's response is not supported by the set of facts, it is likely to contain hallucinations. In such cases, the graph is interrupted and the user has the control of the next step: retry only the last generation step without restarting the entire workflow or end the process. This human-in-the-loop step ensures user control while avoiding unintended loops or undesired actions.

The `interrupt` function in LangGraph enables human-in-the-loop workflows by pausing the graph at a specific node, presenting information to a human, and resuming the graph with their input. This function is useful for tasks like approvals, edits, or collecting additional input. The `interrupt` function is used in conjunction with the `Command` object to resume the graph with a value provided by the human.

```
def human_approval(
    state: AgentState,
):
    _binary_score = state.hallucination.binary_score
    if _binary_score == "1":
        return "END"
    else:
        retry_generation = interrupt(
        {
            "question": "Is this correct?",
            "llm_output": state.messages[-1]
        })

        if retry_generation == "y":
            print("voglio continuare")
            return "respond"
        else:
            return "END"
```

4.3 Building main graph

```
from langgraph.graph import END, START, StateGraph
from langgraph.checkpoint.memory import MemorySaver

checkpointer = MemorySaver()

builder = StateGraph(AgentState, input=InputState)
builder.add_node(analyze_and_route_query)
```

```
    builder.add_edge(START, "analyze_and_route_query")
    builder.add_conditional_edges("analyze_and_route_query", route_query)
    builder.add_node(create_research_plan)
    builder.add_node(ask_for_more_info)
    builder.add_node(respond_to_general_query)
    builder.add_node(conduct_research)
    builder.add_node("respond", respond)
    builder.add_node(check_hallucinations)
    builder.add_conditional_edges("check_hallucinations", human_approval, {"END": E
    builder.add_edge("create_research_plan", "conduct_research")
    builder.add_conditional_edges("conduct_research", check_finished)
    builder.add_edge("respond", "check_hallucinations")

    graph = builder.compile(checkpointer=checkpointer)
```

Building main function (app.py)

"Each function is defined as `async` to enable the streaming behavior during the generation steps.

```
from subgraph.graph_states import ResearcherState
from main_graph.graph_states import AgentState
from utils.utils import config, new_uuid
from subgraph.graph_builder import researcher_graph
from main_graph.graph_builder import InputState, graph
from langgraph.types import Command
import asyncio
import uuid

import asyncio
import time
import builtins

thread = {"configurable": {"thread_id": new_uuid()}}

async def process_query(query):
    inputState = InputState(messages=query)

    async for c, metadata in graph.astream(input=inputState, stream_mode="messa
        if c.additional_kwargs.get("tool_calls"):
            print(c.additional_kwargs.get("tool_calls")[0]["function"].get("arg
        if c.content:
            time.sleep(0.05)
            print(c.content, end="", flush=True)

    if len(graph.get_state(thread)[-1]) > 0:
        if len(graph.get_state(thread)[-1][0].interrupts) > 0:
            response = input("\nThe response may contain uncertain information.
                if response.lower() == 'y':
                    async for c, metadata in graph.astream(Command(resume=response)
                        if c.additional_kwargs.get("tool_calls"):
                            print(c.additional_kwargs.get("tool_calls")[0]["functic
                        if c.content:
                            time.sleep(0.05)
                            print(c.content, end="", flush=True)


async def main():
    input = builtins.input
    print("Enter your query (type '-q' to quit):")
    while True:
        query = input("> ")
        if query.strip().lower() == "-q":
            print("Exiting...")
            break
        await process_query(query)


if __name__ == "__main__":
    asyncio.run(main())
```

After the first invocation, the graph state is checked for interrupts. If any are found, the graph can be invoked again using the command:

```
graph.astream(Command(resume=response), stream_mode="messages", config=thread)
```

In this way, the workflow will resume from the interrupted step without re-executing the previous steps, using the same `thread_id`.

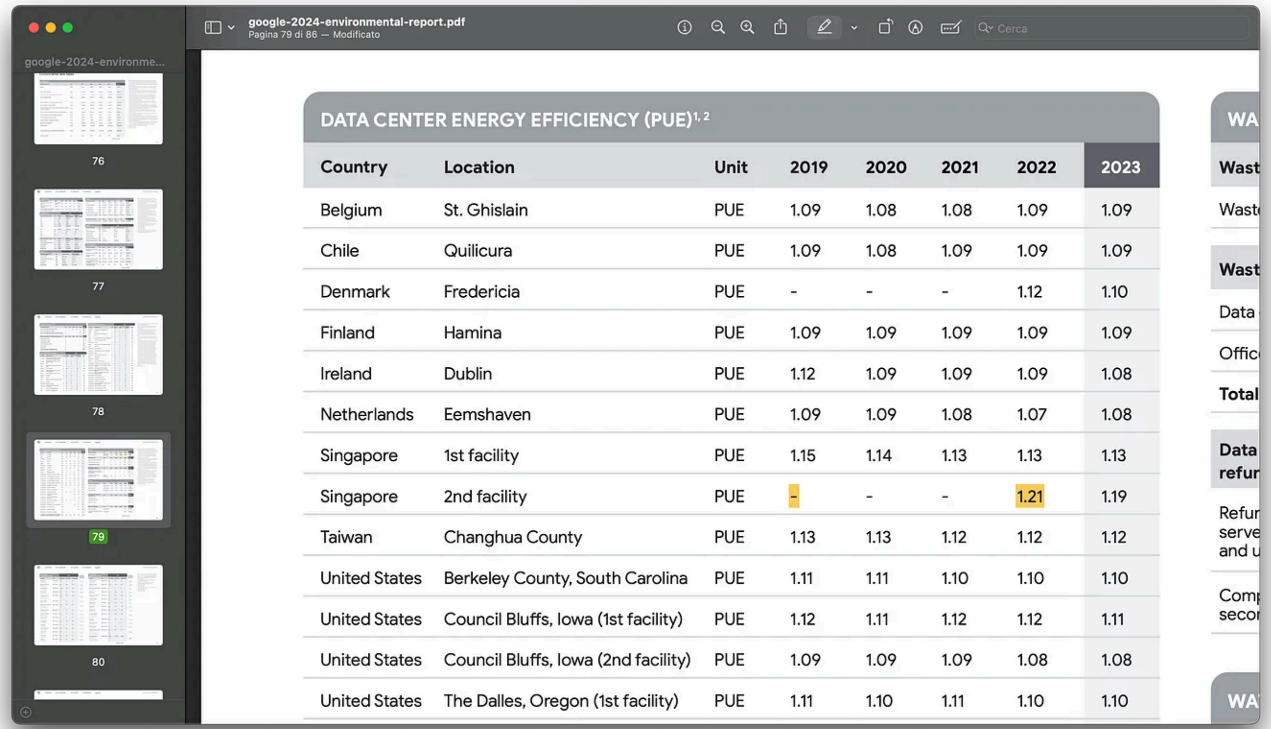
3. Results

For the following test, an annual report on Google's strategy regarding environmental sustainability was used, freely available here.

Live Test

As the first test, the following query was executed to extract different values from different tables, combining the capabilities of a multi-step approach and leveraging the parsing features of the Docling library.

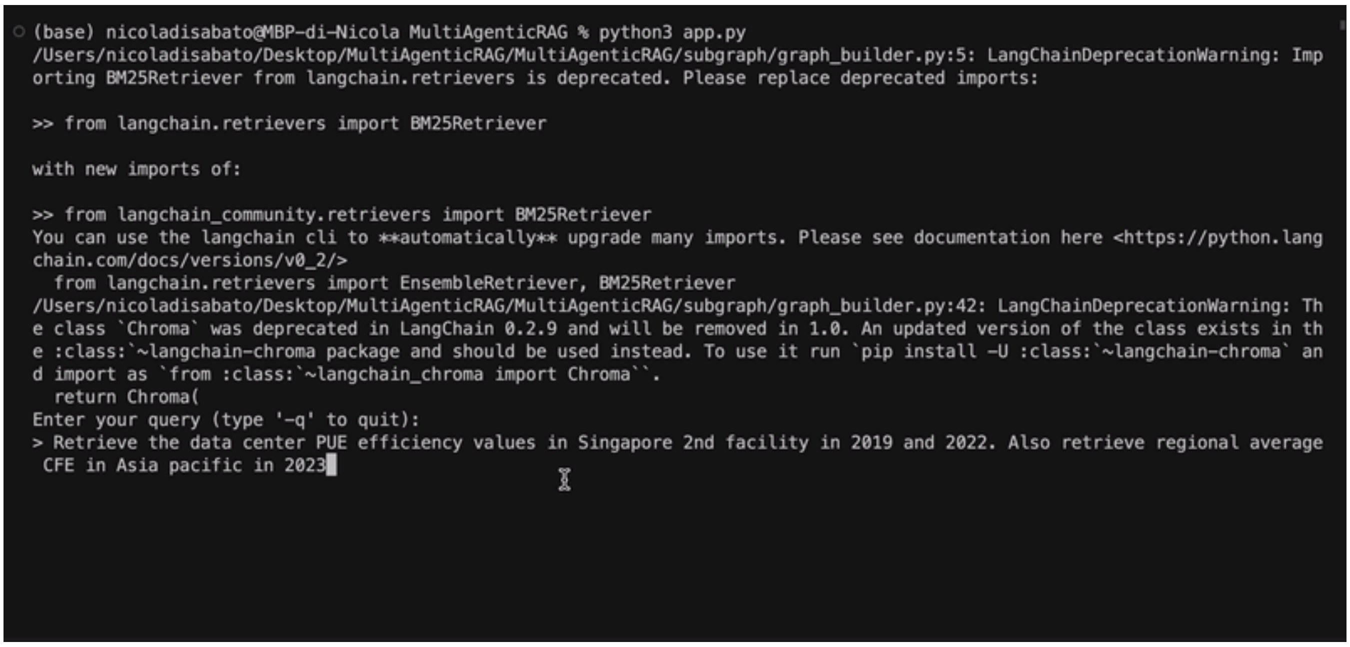
Complex question: *"Retrieve the data center PUE efficiency values in Singapore 2nd facility in 2019 and 2022. Also retrieve regional average CFE in Asia pacific in 2023"*


google-2024-environmental-report.pdf
Pagina 79 di 86 — Modificato

DATA CENTER ENERGY EFFICIENCY (PUE)[1, 2]

Country	Location	Unit	2019	2020	2021	2022	2023
Belgium	St. Ghislain	PUE	1.09	1.08	1.08	1.09	1.09
Chile	Quilicura	PUE	1.09	1.08	1.09	1.09	1.09
Denmark	Fredericia	PUE	-	-	-	1.12	1.10
Finland	Hamina	PUE	1.09	1.09	1.09	1.09	1.09
Ireland	Dublin	PUE	1.12	1.09	1.09	1.09	1.08
Netherlands	Eemshaven	PUE	1.09	1.09	1.08	1.07	1.08
Singapore	1st facility	PUE	1.15	1.14	1.13	1.13	1.13
Singapore	2nd facility	PUE	-	-	-	1.21	1.19
Taiwan	Changhua County	PUE	1.13	1.13	1.12	1.12	1.12
United States	Berkeley County, South Carolina	PUE	1.11	1.11	1.10	1.10	1.10
United States	Council Bluffs, Iowa (1st facility)	PUE	1.12	1.11	1.12	1.12	1.11
United States	Council Bluffs, Iowa (2nd facility)	PUE	1.09	1.09	1.09	1.08	1.08
United States	The Dalles, Oregon (1st facility)	PUE	1.11	1.10	1.11	1.10	1.10

Correct informations


```
(base) nicoladisabato@MBP-di-Nicola MultiAgenticRAG % python3 app.py
/Users/nicoladisabato/Desktop/MultiAgenticRAG/MultiAgenticRAG/subgraph/graph_builder.py:5: LangChainDeprecationWarning: Imp
orting BM25Retriever from langchain.retrievers is deprecated. Please replace deprecated imports:

>> from langchain.retrievers import BM25Retriever

with new imports of:

>> from langchain_community.retrievers import BM25Retriever
You can use the langchain cli to **automatically** upgrade many imports. Please see documentation here <https://python.lang
chain.com/docs/versions/v0_2/>
  from langchain.retrievers import EnsembleRetriever, BM25Retriever
/Users/nicoladisabato/Desktop/MultiAgenticRAG/MultiAgenticRAG/subgraph/graph_builder.py:42: LangChainDeprecationWarning: Th
e class `Chroma` was deprecated in LangChain 0.2.9 and will be removed in 1.0. An updated version of the class exists in th
e :class:`~langchain-chroma package and should be used instead. To use it run `pip install -U :class:`~langchain-chroma` an
d import as `from :class:`~langchain_chroma import Chroma``.
  return Chroma(
Enter your query (type '-q' to quit):
> Retrieve the data center PUE efficiency values in Singapore 2nd facility in 2019 and 2022. Also retrieve regional average
 CFE in Asia pacific in 2023
```

Live demo

The complete result is **correct**, and the **hallucination-check was successfully passed.**

Steps generated by the chatbot:

- "Look up the PUE efficiency values for the Singapore 2nd facility for the years 2019 and 2022.",
- "Find the regional average CFE for the Asia Pacific region in 2023."

Generated text: *"- The Power Usage Effectiveness (PUE) for the Singapore 2nd facility in 2019 is not available, as the data for that year is not provided. However, the PUE for 2022 is 1.21.*

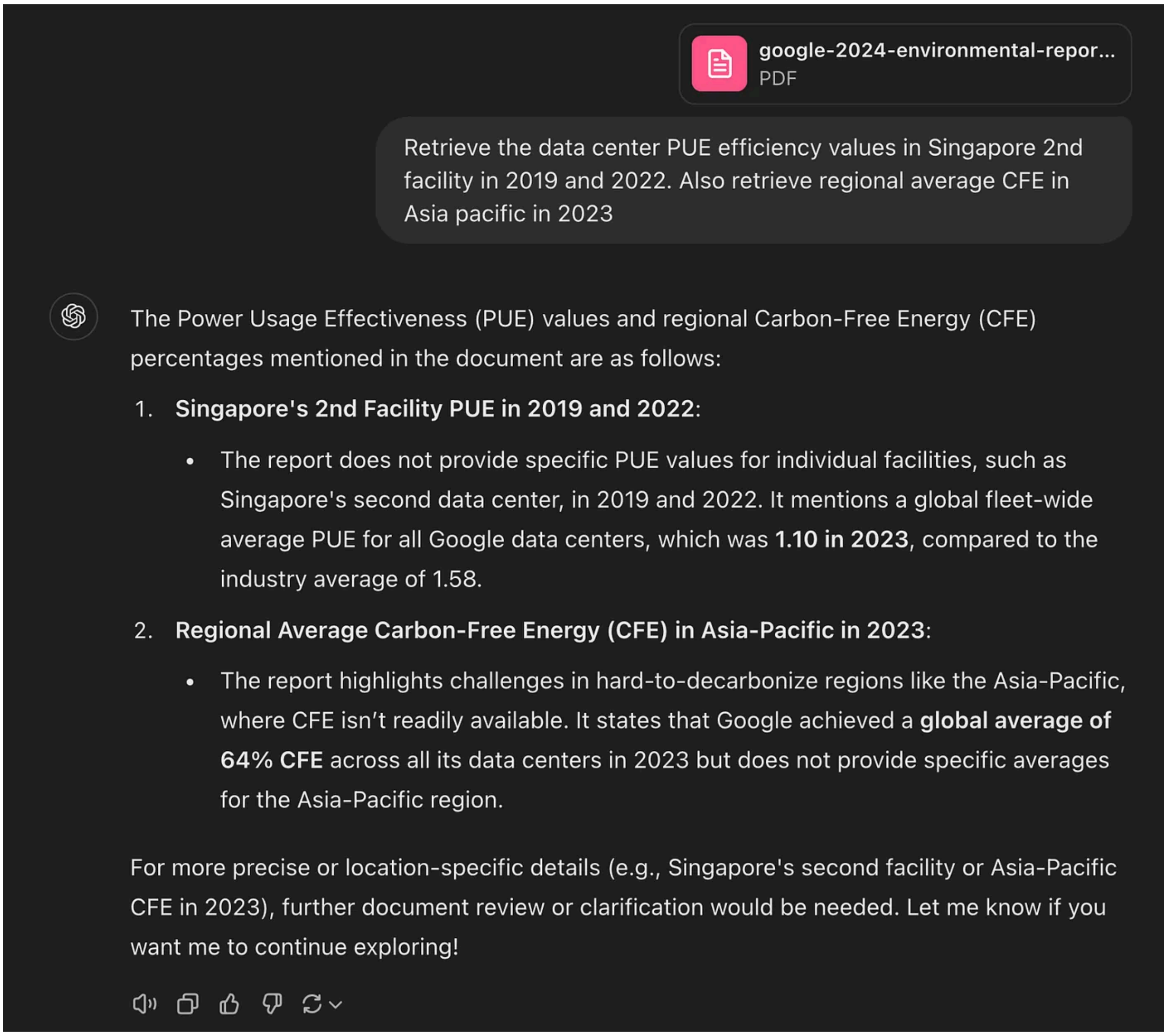
The regional average Carbon-Free Energy (CFE) in the Asia Pacific for 2023 is 12%."

Full output:

```
Enter your query (type '-q' to quit):
> Retrieve the data center PUE efficiency values in Singapore 2nd facility in 2
2025-01-10 20:39:53,381 - INFO - ---ANALYZE AND ROUTE QUERY---
2025-01-10 20:39:53,381 - INFO - MESSAGES: [HumanMessage(content='Retrieve the 
{"logic":"Retrieve the data center PUE efficiency values in Singapore 2nd facil
{"steps":["Look up the PUE efficiency values for the Singapore 2nd facility for
{"queries":["PUE efficiency values Singapore 2nd facility 2019","PUE efficiency
2025-01-10 20:39:58,288 - INFO - ---RETRIEVING DOCUMENTS---
2025-01-10 20:39:58,288 - INFO - Query for the retrieval process: PUE efficienc
2025-01-10 20:39:59,568 - INFO - ---RETRIEVING DOCUMENTS---
2025-01-10 20:39:59,568 - INFO - Query for the retrieval process: PUE efficienc
2025-01-10 20:40:00,891 - INFO - ---RETRIEVING DOCUMENTS---
2025-01-10 20:40:00,891 - INFO - Query for the retrieval process: Look up the P
2025-01-10 20:40:01,820 - INFO - 
4 documents retrieved in total for the step: Look up the PUE efficiency values 
2025-01-10 20:40:01,825 - INFO - ---GENERATE QUERIES---
{"queries":["Asia Pacific regional average CFE 2023","CFE statistics Asia Pacif
2025-01-10 20:40:02,780 - INFO - ---RETRIEVING DOCUMENTS---
2025-01-10 20:40:02,780 - INFO - Query for the retrieval process: Asia Pacific 
2025-01-10 20:40:03,757 - INFO - ---RETRIEVING DOCUMENTS---
2025-01-10 20:40:03,757 - INFO - Query for the retrieval process: CFE statistic
2025-01-10 20:40:04,885 - INFO - ---RETRIEVING DOCUMENTS---
2025-01-10 20:40:04,885 - INFO - Query for the retrieval process: Find the regi
2025-01-10 20:40:06,526 - INFO - 
4 documents retrieved in total for the step: Find the regional average CFE for 
2025-01-10 20:40:06,530 - INFO - --- RESPONSE GENERATION STEP ---
- The Power Usage Effectiveness (PUE) for the Singapore 2nd facility in 2019 is

- The regional average Carbon-Free Energy (CFE) in the Asia Pacific for 2023 is
{"binary_score":"1"}>
```

Now let's try it on **ChatGPT**. After uploading the pdf file to the web app, the same query was made.

As shown in the image, the values returned by **ChatGPT** are incorrect, and the model exhibited hallucinations. In this case, a hallucination check step would have allowed the response to be regenerated (Self-Reflective RAG).



google-2024-environmental-repor...
PDF

Retrieve the data center PUE efficiency values in Singapore 2nd facility in 2019 and 2022. Also retrieve regional average CFE in Asia pacific in 2023

The Power Usage Effectiveness (PUE) values and regional Carbon-Free Energy (CFE) percentages mentioned in the document are as follows:

1. **Singapore's 2nd Facility PUE in 2019 and 2022**:
   - The report does not provide specific PUE values for individual facilities, such as Singapore's second data center, in 2019 and 2022. It mentions a global fleet-wide average PUE for all Google data centers, which was **1.10 in 2023**, compared to the industry average of 1.58.
2. **Regional Average Carbon-Free Energy (CFE) in Asia-Pacific in 2023**:
   - The report highlights challenges in hard-to-decarbonize regions like the Asia-Pacific, where CFE isn't readily available. It states that Google achieved a **global average of 64% CFE** across all its data centers in 2023 but does not provide specific averages for the Asia-Pacific region.

For more precise or location-specific details (e.g., Singapore's second facility or Asia-Pacific CFE in 2023), further document review or clarification would be needed. Let me know if you want me to continue exploring!

ChatGPT test

4. Conclusion

Agentic RAG: Technical Challenges and Considerations

Despite the improved performance, implementing Agentic RAG is not without its challenges:

- **Latency:** The increased complexity of agentic interactions often leads to longer response times. Striking a balance between speed and accuracy is a critical