Quantized_LLM_Agentic_RAG_Notebook

August 30, 2025

1 Deep Dive Agentic Retrieval Augmented Generation

An Agentic RAG is required when we use reasoning to determine which action(s) to take and in which order to take them. Essentially we use agents instead of a LLM directly to accomplish a set of tasks which requires planning, multi step reasoning, tool use and/or learning over time. Agents give us agency!

Agency : The ability to take action or to choose what action to take

In the context of RAG, we can plug in agents to enhance the reasoning prior to selection of RAG pipelines, within a RAG pipeline for retrieval or reranking and finally for synthesising before we send out the response. This improves RAG to a large extent by automating complex workflows and decisions that are required for a non trivial RAG use case.

1.0.1 Purpose of this Agentic RAG

This notebook presents a practical implementation of Agentic Retrieval-Augmented Generation (RAG)—a system where decision-making and tool selection are delegated to an intelligent agent before executing a response. Rather than passing every query through a static RAG pipeline, this system introduces agency—the ability to choose the best course of action depending on the nature of the query.

At the heart of this implementation is a router prompt, which classifies user queries into one of three categories:

- OpenAI documentation: Queries related to tools, APIs, or usage guidelines for OpenAI models
- 10-K financial reports: Questions requiring retrieval from company filings or financial datasets
- Live Internet search: Broader, current, or comparative queries that need web access

Once the query is classified, the system invokes a corresponding route handler:

- For OpenAI and 10-K queries, it retrieves relevant context from a vector database (Qdrant) using text embeddings, then applies a RAG-based response generator.
- For Internet queries, it fetches real-time information using a web-access API (ARES).

This approach is an example of Agentic RAG, where reasoning precedes retrieval and generation. By plugging in agents before and within the RAG pipeline, we make the system smarter and more adaptive. This allows us to:

- Automatically choose the right retrieval method based on context
- Combine structured knowledge with real-time search

- Scale RAG beyond trivial use cases by integrating multi-step decision logic

Importantly, no external agentic frameworks are used—this is a ground-up implementation that demonstrates how to build a lightweight but intelligent agentic system using only a language model, prompt engineering, and retrieval tools.

1.1 Setup and Dependencies

[1]:
```
# Install the necessary libraries
#!pip install openai
#!pip install qdrant_client
#!pip install transformers
#!pip install -q -U bitsandbytes
from dotenv import load_dotenv
import os
load_dotenv()
```

[1]:
```
True
```

[2]:
```
# Import basic libraries
import requests             # Used for making HTTP requests (e.g., calling ARES␣
 ↪API for live internet queries)
import json                 # For parsing and structuring JSON data (especially␣
 ↪OpenAI and routing responses)

# Google Colab-specific (for securely handling API keys)
#from google.colab import userdata  # To securely store and retrieve␣
 ↪credentials in Colab

# OS operations
import os                   # Useful for accessing environment variables and␣
 ↪managing paths

# OpenAI API client
from openai import OpenAI   # Official OpenAI client library to interface with␣
 ↪GPT models for routing and generation

# Text processing
import re                   # Regular expressions for cleaning or preprocessing␣
 ↪inputs (if needed)

# Optional visualization (for analysis/debugging purposes)
import matplotlib.pyplot as plt       # For displaying charts or visual debug␣
 ↪outputs (e.g., embeddings visualizations)
import matplotlib.image as mpimg      # For loading/displaying images if needed␣
 ↪(rare in RAG, but helpful in demos)
```

```
# Embedding models (used for text vectorization during retrieval)
from transformers import AutoTokenizer, AutoModel  # For loading custom␣
 ↪transformer models if not using OpenAI embeddings

# Vector database client
from qdrant_client import QdrantClient   # Qdrant is used as the vector store␣
 ↪to retrieve documents based on similarity
```

1.2 1. Defining the Internet Tool

First, we will define a tool function that enables our system to answer queries requiring real-time, internet-based information. Not all questions can be answered using static documents like OpenAI docs or financial filings—sometimes users ask about current trends, comparisons, or live updates.

To handle this, we introduce a live search capability using the **ARES API** by Traversaal.

1.2.1 What is ARES API?

ARES is a web-based tool that allows you to:

- Search the internet in real time.
- Get LLM-generated answers based on live search results.

This is particularly useful for questions about:

- Current events (e.g., *"Latest AI tools in 2025"*),
- Tech comparisons (e.g., *"Gemini vs GPT-4"*),
- General knowledge outside internal datasets.

Please generate the API key here

[3]:
```
#loads ares api key from colab secrets
ares_api_key=os.getenv('ARES_API_KEY')
```

[4]:
```
import requests  # For sending HTTP POST requests to the ARES API

def get_internet_content(user_query: str, action: str):
    """
    Fetches a response from the internet using ARES-API based on the user's␣
 ↪query.

    This function serves as the tool invoked when the router classifies a query
    as requiring real-time information beyond internal datasets-i.e.,␣
 ↪"INTERNET_QUERY".
    It sends the query to a live search API (ARES) and returns the result.

    Args:
        user_query (str): The user's question that needs a live answer.
        action (str): Route type (always expected to be "INTERNET_QUERY").
```

```
    Returns:
        str: Response text generated using internet search or an error message.
    """
    print("Getting your response from the internet   ...")

    # API endpoint for the ARES live search tool
    url = "https://api-ares.traversaal.ai/live/predict"

    # Payload structure expected by the ARES API
    payload = {"query": [user_query]}

    # Authentication and content headers for API access
    headers = {
        "x-api-key": ares_api_key,  # Your secret API key (should be securely␣
↪loaded from environment)
        "content-type": "application/json"
    }

    try:
        # Send the query to the ARES API and check for success
        response = requests.post(url, json=payload, headers=headers)
        response.raise_for_status()

        # Extract and return the main response text from the API's nested JSON
        return response.json().get('data', {}).get('response_text', "No␣
↪response received.")

    # Handle HTTP-level errors (e.g., 400s or 500s)
    except requests.exceptions.HTTPError as http_err:
        return f"HTTP error occurred: {http_err}"

    # Handle general connection, timeout, or request formatting issues
    except requests.exceptions.RequestException as req_err:
        return f"Request error occurred: {req_err}"

    # Catch-all for any unexpected failure
    except Exception as err:
        return f"An unexpected error occurred: {err}"
```

[5]:
```
print(get_internet_content("Tell me about best travel destinations in 2025?
↪","INTERNET_QUERY")) #run internet function to test results
```

```
Getting your response from the internet   …
**Best Travel Destinations in 2025**

Here are some of the top travel destinations in 2025:

1. **Western Colorado, USA**: Experience ranch life and ecological renewal in
```

this beautiful region.
2. **Byron Bay, Australia**: Enjoy the stunning beaches and laid-back atmosphere of this popular destination.
3. **Mendoza, Argentina**: Visit this northern region of Argentina in winter for a lovely experience.
4. **Bali, Indonesia**: Explore the beautiful beaches, temples, and culture of this island paradise.
5. **Dubai, United Arab Emirates**: Discover the luxurious shopping, dining, and entertainment options of this modern city.
6. **Sicily, Italy**: Visit the ancient ruins, beautiful beaches, and vibrant culture of this Italian island.
7. **Paris, France**: Explore the City of Light's famous landmarks, art museums, and romantic atmosphere.
8. **London, England**: Discover the British capital's iconic landmarks, cultural attractions, and vibrant neighborhoods.
9. **Ahr Valley, Germany**: Visit this scenic region for its beautiful vineyards, castles, and traditional German cuisine.
10. **Alaska, USA**: Experience the breathtaking natural beauty of America's largest state, with its vast wilderness, glaciers, and wildlife.
11. **Cuba**: Explore the vibrant culture, beautiful beaches, and historic landmarks of this Caribbean island.
12. **Djerba, Tunisia**: Visit this North African island for its beautiful beaches, ancient ruins, and traditional Berber culture.
13. **Providence, Rhode Island, USA**: Discover the charming historic district, vibrant arts scene, and delicious seafood of this New England city.
14. **Maine, USA**: Visit the rugged coastline, picturesque towns, and delicious lobster of this northeastern state.
15. **Santa Barbara, California, USA**: Enjoy the beautiful beaches, wine country, and Spanish architecture of this charming coastal town.
16. **Seattle, Washington, USA**: Explore the vibrant music scene, coffee culture, and stunning natural beauty of this Pacific Northwest city.
17. **Pennsylvania, USA**: Visit the historic cities, scenic countryside, and world-class museums of this eastern state.
18. **Nebraska, USA**: Discover the scenic Great Plains, historic sites, and vibrant arts scene of this Midwestern state.
19. **Zanzibar, Tanzania**: Enjoy the beautiful beaches, historic Stone Town, and vibrant culture of this Indian Ocean island.
20. **Kenya**: Experience the thrilling safaris, stunning natural beauty, and vibrant culture of this East African country.
21. **Scotland**: Visit the rugged Highlands, historic castles, and vibrant culture of this northern UK country.
22. **Spain**: Explore the beautiful beaches, historic cities, and vibrant culture of this southern European country.
23. **France**: Visit the famous landmarks, art museums, and romantic atmosphere of this western European country.
24. **Mexico**: Discover the vibrant culture, beautiful beaches, and historic landmarks of this Central American country.
25. **Thailand**: Enjoy the beautiful beaches, delicious cuisine, and vibrant

```
culture of this Southeast Asian country.

These are just a few of the many amazing travel destinations in 2025. Whether
you're looking for adventure, culture, or relaxation, there's something for
everyone.
```

1.3 2. Router Query Function — Giving the Agent Its Brain

In this step, we will define the router function, which plays a critical role in our Agentic RAG system.

1.3.1 What is a Router?

A router is like the decision-making brain of our assistant.

Before trying to answer a user's question, the system first needs to figure out:

> "Where should I go to find the right answer?"

To make this decision, we use the OpenAI GPT model. We provide it with a detailed system prompt that explains how to classify the user's question into one of these categories:

- **OPENAI_QUERY** → Questions about OpenAI tools, APIs, models, or documentation.
- **10K_DOCUMENT_QUERY** → Questions about companies, financial filings, or analysis based on 10-K reports.
- **INTERNET_QUERY** → Anything else that likely requires real-time or general web information.

1.3.2 What does the function do?

- Sends the user's question to the OpenAI API.
- Receives a JSON response containing:
  - `action`: The category the query belongs to.
  - `reason`: A short explanation for the decision.
  - `answer`: (Optional) A quick response if it's simple enough (left blank for internet queries).
- Parses the response and returns it as a Python dictionary.

1.3.3 Why is this important?

This router gives the system agency—the ability to decide which knowledge source to use. It's what makes this pipeline agentic, not just static.

Without the router, every query would follow the same path. With it, we can:

- Dynamically switch between tools and data sources.
- Handle different types of user questions intelligently.
- Avoid wasting resources on unnecessary steps.

[6]:
```python
# Securely retrieve the OpenAI API key from Colab's user data store
# This avoids hardcoding sensitive credentials directly in the notebook
openai_api_key = os.getenv('OPENAI_API_KEY')
```

```
# Initialize the OpenAI client with the retrieved API key
# This client will be used for:
# - Query classification via the router prompt
# - Potentially generating responses from retrieved context
openaiclient = OpenAI(api_key=openai_api_key)
```

[7]:
```
from openai import OpenAIError

def route_query(user_query: str):
    router_system_prompt =f"""
    As a professional query router, your objective is to correctly classify␣
 ↪user input into one of three categories based on the source most relevant␣
 ↪for answering the query:
    1. "OPENAI_QUERY": If the user's query appears to be answerable using␣
 ↪information from OpenAI's official documentation, tools, models, APIs, or␣
 ↪services (e.g., GPT, ChatGPT, embeddings, moderation API, usage guidelines).
    2. "10K_DOCUMENT_QUERY": If the user's query pertains to a collection of␣
 ↪documents from the 10k annual reports, datasets, or other structured␣
 ↪documents, typically for research, analysis, or financial content.
    3. "INTERNET_QUERY": If the query is neither related to OpenAI nor the 10k␣
 ↪documents specifically, or if the information might require a broader search␣
 ↪(e.g., news, trends, tools outside these platforms), route it here.

    Your decision should be made by assessing the domain of the query.

    Always respond in this valid JSON format:
    {{
        "action": "OPENAI_QUERY" or "10K_DOCUMENT_QUERY" or "INTERNET_QUERY",
        "reason": "brief justification",
        "answer": "AT MAX 5 words answer. Leave empty if INTERNET_QUERY"
    }}

    EXAMPLES:

    - User: "How to fine-tune GPT-3?"
    Response:
    {{
        "action": "OPENAI_QUERY",
        "reason": "Fine-tuning is OpenAI-specific",
        "answer": "Use fine-tuning API"
    }}

    - User: "Where can I find the latest financial reports for the last 10␣
 ↪years?"
    Response:
    {{
```

```
        "action": "10K_DOCUMENT_QUERY",
        "reason": "Query related to annual reports",
        "answer": "Access through document database"
    }}

    - User: "Top leadership styles in 2024"
    Response:
    {{
        "action": "INTERNET_QUERY",
        "reason": "Needs current leadership trends",
        "answer": ""
    }}

    - User: "What's the difference between ChatGPT and Claude?"
    Response:
    {{
        "action": "INTERNET_QUERY",
        "reason": "Cross-comparison of different providers",
        "answer": ""
    }}

    Strictly follow this format for every query, and never deviate.
    User: {user_query}
    """

    try:
        # Query the GPT-4 model with the router prompt and user input
        response = openaiclient.chat.completions.create(
            model="gpt-4o",
            messages=[{"role": "system", "content": router_system_prompt}]
        )

        # Extract and parse the model's JSON response
        task_response = response.choices[0].message.content
        json_match = re.search(r"\{.*\}", task_response, re.DOTALL)
        json_text = json_match.group()
        parsed_response = json.loads(json_text)
        return parsed_response

    # Handle OpenAI API errors (e.g., rate limits, authentication)
    except OpenAIError as api_err:
        return {
            "action": "INTERNET_QUERY",
            "reason": f"OpenAI API error: {api_err}",
            "answer": ""
        }
```

```
    # Handle case where model response isn't valid JSON
    except json.JSONDecodeError as json_err:
        return {
            "action": "INTERNET_QUERY",
            "reason": f"JSON parsing error: {json_err}",
            "answer": ""
        }

    # Catch-all for any other unforeseen issues
    except Exception as err:
        return {
            "action": "INTERNET_QUERY",
            "reason": f"Unexpected error: {err}",
            "answer": ""
        }
```

[8]:
```
route_query("what is the revenue of uber in 2021?")
```

[8]:
```
{'action': 'INTERNET_QUERY',
 'reason': 'Requires specific company data',
 'answer': ''}
```

1.4 3. Setting Up Qdrant Vector Database for Agentic RAG

In this step, we are connecting our agent to a pre-built vector database using Qdrant—a tool used to store and search document embeddings (numerical representations of text).

What Are We Doing? We are loading an existing Qdrant database that was downloaded from a GitHub repository. This database already contains:

- Vectorized OpenAI documentation
- Vectorized 10-K financial filings

By loading this saved data:

- We save time (no need to re-embed the documents)
- We enable fast similarity search to retrieve relevant text chunks

This setup allows our system to perform semantic search, meaning it can understand the meaning of the user query and match it with the most relevant pieces of information stored in the database.

1.4.1 Why This Matters in Agentic RAG

Once the router decides that the query should go to the OpenAI docs or the 10-K reports, our system uses Qdrant to:

- Search for the most relevant pieces of text
- Pass those to the model to generate a grounded answer

So, this step is essential to support retrieval-augmented generation (RAG) within our agentic flow.

#Data Sources:

10K Database: Lyft 2024 & Uber 2021 SEC filings

OpenAI Docs: Official OpenAI documentation

For lecture demo purposes, the vecitr database has already been created and hosted on Github which we will clone here. In order to create your own embeddings, the notebook and data will be hosted and shared on github

[9]:
```
# Clone the project repository that contains prebuilt vector data (e.g., Qdrant␣
 ↪collections)
# This includes document embeddings and configurations needed for retrieval␣
 ↪(10-K, OpenAI docs)
!git clone https://github.com/hamzafarooq/multi-agent-course.git
```

```
fatal: destination path 'multi-agent-course' already exists and is not an empty
directory.
```

[10]:
```
#  Initializing Qdrant client with local path to vector database
# The path points to prebuilt Qdrant collections (10-K and OpenAI docs) cloned␣
 ↪from the repository
# This enables fast, local retrieval of relevant document chunks based on␣
 ↪semantic similarity
client = QdrantClient(path="multi-agent-course/Module_1/Agentic_RAG/
 ↪qdrant_data")
```

1.5 4. Building the Retriever and RAG for Vector Databases

In this section, we build the core logic that allows our agent to find relevant documents and generate grounded answers using them.

###Step 1: Import the Embedding Model We start by importing the nomic-ai/nomic-embed-text-v1.5 model from Hugging Face. This model is used to convert any text (such as a user query) into a dense vector, known as an embedding. These embeddings capture the semantic meaning of text, allowing us to later compare and retrieve similar documents.

[11]:
```
# Load the tokenizer and embedding model from Hugging Face
# This model converts raw text into dense vector representations (embeddings)
# Used for similarity search in Qdrant during document retrieval
text_tokenizer = AutoTokenizer.from_pretrained("nomic-ai/nomic-embed-text-v1.
 ↪5", trust_remote_code=True)
text_model = AutoModel.from_pretrained("nomic-ai/nomic-embed-text-v1.5",␣
 ↪trust_remote_code=True)

def get_text_embeddings(text):
    """
    Converts input text into a dense embedding using the Nomic embedding model.
    These embeddings are used to query Qdrant for semantically relevant␣
 ↪document chunks.
```

```
    Args:
        text (str): The input text or query from the user.

    Returns:
        np.ndarray: A fixed-size vector representing the semantic meaning of␣
↪the input.
    """
    # Tokenize and prepare input for the model
    inputs = text_tokenizer(text, return_tensors="pt", padding=True,␣
↪truncation=True)

    # Forward pass to get model outputs
    outputs = text_model(**inputs)

    # Take the mean across all token embeddings to get a single vector (pooled␣
↪representation)
    embeddings = outputs.last_hidden_state.mean(dim=1)

    # Convert to NumPy array and detach from computation graph
    return embeddings[0].detach().numpy()

# Example usage: Generate and preview the embedding of a test sentence
text = "This is a test sentence."
embeddings = get_text_embeddings(text)
print(embeddings[:5])  # Print first 5 dimensions for inspection
```

```
<All keys matched successfully>
```

```
[ 1.2799689   0.40158418 -3.5162663  -0.3981327   1.5919126 ]
```

1.5.1 Step 2: Define the Embedding Function

We then define a function get_text_embeddings() which:

- Tokenizes the input text
- Runs it through the model
- Computes the average of all token embeddings
- Returns a single vector that represents the full sentence

This vector will be used to query Qdrant to find the most relevant document chunks based on similarity.

[12]:
```
def rag_formatted_response(user_query: str, context: list):
    """
    Generate a response to the user query using the provided context,
    with article references formatted as [1][2], etc.

    This function performs the final step in the RAG pipeline-synthesizing an␣
↪answer
```

```
    from retrieved document chunks (context). It prompts the model to generate a
    grounded response, explicitly citing sources using a reference format.

    Args:
        user_query (str): The user's original question.
        context (list): List of text chunks retrieved from Qdrant (10-K or␣
↪OpenAI docs).

    Returns:
        str: A generated response grounded in the retrieved context, with␣
↪numbered citations.
    """

    # Construct a RAG prompt that includes both:
    # 1. The user's query
    # 2. The supporting context documents
    # The prompt instructs the model to answer using only the provided context,
    # and to include citations like [1], [2], etc. based on chunk IDs or order.
    rag_prompt = f"""
       Based on the given context, answer the user query: {user_query}\nContext:
↪\n{context}
       and employ references to the ID of articles provided [ID], ensuring␣
↪their relevance to the query.
       The referencing should always be in the format of [1][2]... etc. </
↪instructions>
    """

    #  Call GPT-4o to generate the response using the RAG-style prompt
    response = openaiclient.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": rag_prompt},
        ]
    )

    # Return the model's generated answer
    return response.choices[0].message.content
```

1.5.2 Step 3: Define the RAG Response Generator

After retrieving relevant text chunks from Qdrant, we use the rag_formatted_response() function to generate a final answer. This function:

- Takes the user query and the retrieved document chunks
- Builds a prompt that asks the language model (GPT-4o) to answer the question using only the provided context
- Instructs the model to include references like [1], [2] for traceability

This ensures the output is not only informative but also grounded in actual retrieved data.

Together, these two functions lay the foundation for combining retrieval (from vector DB) and generation (from LLM) — the two pillars of a RAG system.

[13]:
```
def retrieve_and_response(user_query: str, action: str):
    """
    Retrieves relevant text chunks from the appropriate Qdrant collection
    based on the query type, then generates a response using RAG.

    This function powers the retrieval and response generation pipeline
    for queries that are classified as either OPENAI-related or 10-K related.
    It uses semantic search to fetch relevant context from a Qdrant vector store
    and then generates a response using that context via a RAG prompt.

    Args:
        user_query (str): The user's input question.
        action (str): The classification label from the router (e.g.,␣
↪"OPENAI_QUERY", "10K_DOCUMENT_QUERY").

    Returns:
        str: A model-generated response grounded in retrieved documents, or an␣
↪error message.
    """

    # Define mapping of routing labels to their respective Qdrant collections
    collections = {
        "OPENAI_QUERY": "opnai_data",           # Collection of OpenAI␣
↪documentation embeddings
        "10K_DOCUMENT_QUERY": "10k_data"        # Collection of 10-K financial␣
↪document embeddings
    }

    try:
        # Ensure that the provided action is valid
        if action not in collections:
            return "Invalid action type for retrieval."

        # Step 1: Convert the user query into a dense vector (embedding)
        try:
            query = get_text_embeddings(user_query)
        except Exception as embed_err:
            return f"Embedding error: {embed_err}"  # Fail early if embedding␣
↪fails

        # Step 2: Retrieve top-matching chunks from the relevant Qdrant␣
↪collection
```

```
        try:
            text_hits = client.query_points(
                collection_name=collections[action],  # Choose the right␣
↪collection based on routing
                query=query,                          # The embedding of the␣
↪user's query
                limit=3                               # Fetch top 3 relevant␣
↪chunks
            ).points
        except Exception as qdrant_err:
            return f"Vector DB query error: {qdrant_err}"  # Handle Qdrant␣
↪access issues

        # Extract the raw content from the retrieved vector hits
        contents = [point.payload['content'] for point in text_hits]

        # If no relevant content is found, return early
        if not contents:
            return "No relevant content found in the database."

        # Step 3: Pass the retrieved context to the RAG model to generate a␣
↪response
        try:
            response = rag_formatted_response(user_query, contents)
            return response
        except Exception as rag_err:
            return f"RAG response error: {rag_err}"  # Handle generation␣
↪failures

    # Catch any unforeseen errors in the overall process
    except Exception as err:
        return f"Unexpected error: {err}"
```

2 5. Putting It All Together: Running the Agentic RAG

In this final step, we combine everything into a single function that controls the entire Agentic RAG workflow. The agentic_rag() function acts as the main orchestrator of the system.

Here's what it does:

- Prints the user's query for reference.
- Uses the router function (powered by GPT) to decide which type of data source to use:
  - OpenAI documentation
  - 10-K financial reports
- Internet search
- Calls the correct function based on the route:
- If it's an OpenAI or 10-K query, it retrieves data from Qdrant and generates a RAG response.

- If it's an Internet query, it uses the ARES API to fetch live information.
- Displays the final response, neatly formatted in the console.

This step brings the agentic loop full circle—from understanding the question, reasoning about where to search, to finally responding with the best possible answer.

[14]:
```
# Dictionary that maps the route labels (decided by the router) to their␣
 ↪respective functions
# Each type of query is handled differently:
# - OPENAI_QUERY and 10K_DOCUMENT_QUERY use document retrieval + RAG
# - INTERNET_QUERY uses a web search API
routes = {
    "OPENAI_QUERY": retrieve_and_response,
    "10K_DOCUMENT_QUERY": retrieve_and_response,
    "INTERNET_QUERY": get_internet_content,
}

def agentic_rag(user_query: str):
    """
    Main function that runs the full Agentic RAG system.

    This function takes a user's question, decides what type of query it is␣
 ↪(OpenAI-related,
    financial document-related, or general internet), and then calls the right␣
 ↪function
    to handle it. Finally, it prints out the full conversation and response.

    Args:
        user_query (str): The user's input question.

    Returns:
        None (It just prints the result nicely to the console)
    """

    #  Terminal color codes to make the printed output easier to read and␣
 ↪visually structured
    CYAN = "\033[96m"
    GREY = "\033[90m"
    BOLD = "\033[1m"
    RESET = "\033[0m"

    try:
        # Step 1: Print the user's original question to the console
        print(f"{BOLD}{CYAN} User Query:{RESET} {user_query}\n")

        # Step 2: Use the router (powered by GPT) to decide which route the␣
 ↪query belongs to
```

```
        try:
            response = route_query(user_query)
        except Exception as route_err:
            # If something goes wrong while classifying the query, show an␣
↪error message
            print(f"{BOLD}{CYAN} BOT RESPONSE:{RESET}\n")
            print(f"Routing error: {route_err}\n")
            return

        # Extract the routing decision and the reason behind it
        action = response.get("action")  # e.g., "OPENAI_QUERY"
        reason = response.get("reason")  # e.g., "Related to OpenAI tools"

        # Step 3: Show the selected route and why it was chosen
        print(f"{GREY} Selected Route: {action}")
        print(f" Reason: {reason}")
        print(f" Processing query...{RESET}\n")

        # Step 4: Call the correct function depending on the route (retrieval␣
↪or web search)
        try:
            route_function = routes.get(action)  # Find the function to use for␣
↪this route
            if route_function:
                result = route_function(user_query, action)  # Run the function␣
↪with the user's input
            else:
                result = f"Unsupported action: {action}"  # Catch unknown␣
↪routing types
        except Exception as exec_err:
            result = f"Execution error: {exec_err}"  # Handle failure in the␣
↪chosen route function

        # Step 5: Print the final response to the user
        print(f"{BOLD}{CYAN} BOT RESPONSE:{RESET}\n")
        print(f"{result}\n")

    except Exception as err:
        # Catch-all for any unexpected errors in the overall logic
        print(f"{BOLD}{CYAN} BOT RESPONSE:{RESET}\n")
        print(f"Unexpected error occurred: {err}\n")
```

[15]:
```
agentic_rag("what was uber revenue in 2021?")
```

```
User Query: what was uber revenue in 2021?
```

```
  Selected Route: 10K_DOCUMENT_QUERY

  Reason: Annual revenue found in 10k reports

  Processing query…

  BOT RESPONSE:

Uber's revenue in 2021 was $17,455 million [1].
```

[16]:
```
agentic_rag("what was lyft revenue in 2024?")
```

```
  User Query: what was lyft revenue in 2024?

  Selected Route: INTERNET_QUERY

  Reason: Requires post-2023 financial data

  Processing query…

Getting your response from the internet   …
  BOT RESPONSE:

Lyft revenue in 2024 was $5.8 billion, a 31% year-over-year increase.
```

[17]:
```
agentic_rag("List me down new LLMs in 2025")
```

```
  User Query: List me down new LLMs in 2025

  Selected Route: INTERNET_QUERY

  Reason: Future predictions require broader search

  Processing query…

Getting your response from the internet   …
  BOT RESPONSE:

**New Large Language Models 2025**

Here are the new Large Language Models (LLMs) in 2025:

1. GPT-4.5 (Orion)
2. Claude 3.7 Sonnet
3. Gemini 2.5 Pro
4. DeepSeek-V3-0324
5. Grok-3
6. Qwen3
7. Llama 4
```

8. LLaMA 3 (Meta)
9. Claude Sonnet 4 (Anthropic)
10. Apple's On-Device and Server Foundation Language Models (Apple Intelligence Foundation)

These models are mentioned in various articles and research papers, including:

- "Top LLMs To Use in 2025: Our Best Picks" by Splunk
- "Top 10 open source LLMs for 2025" by Instaclustr
- "Top 9 Large Language Models as of August 2025" by Shakudo
- "Top 10 LLMs Summer 2025 Edition" by Azumo
- "The 10 Best Large Language Models (LLMs) in 2025" by Botpress
- "A List of Large Language Models" by IBM
- "Large Language Models: What You Need to Know in 2025" by Hatchworks
- "Apple Intelligence Foundation Language Models Tech Report 2025" by Apple Machine Learning

[18]:
```
agentic_rag("how to work with chat completions?")
```

User Query: how to work with chat completions?

Selected Route: OPENAI_QUERY

Reason: Chat completions related to OpenAI

Processing query…

BOT RESPONSE:

To work with Chat Completions via OpenAI API, you first need to send a request to the OpenAI API using the endpoint https://api.openai.com/v1/chat/completions. You can retrieve the chat completions using the model specified and further filtered by metadata keys if necessary [1].

If you have a specific chat completion ID that you want to fetch, you can retrieve it and its associated messages by adding {completion_id}/messages in the endpoint. It can only be returned if it was stored with the 'store' parameter set to 'true' [1].

You can see an example of a request with "curl" command below:

```
curl https://api.openai.com/v1/chat/completions/chat_abc123/messages -H "Authorization: Bearer $OPENAI_API_KEY" -H "Content-Type: application/json"
```

The response of this request will be in JSON format, including a list of messages related to the chat completion specified.

```
For working with streamed chat completions in real-time, they offer the ability
to receive chunks of completions returned from the model using server-sent
events [1].

Don't forget that for pagination, consider the 'after', 'limit', and 'order'
parameters in your request [1].
Always ensure that the identifiers for last message or last chat completion are
provided as per the requirement.

Please note, all these requests will need to be authorized with your OpenAI API
key.
```

3 6. CHANGE TO QUANTIZED LLM

In previous assignment we created a quantized model and placed it into Hugging Face. We will use this to rerun the above queries. We will use gpt-neox-20b as the quantized LLM. We will rewrite the route_query to route_query_quantized

[19]:
```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig

# Define the model ID for the large 20B parameter model.
model_id = "EleutherAI/gpt-neox-20b"

# Create a BitsAndBytesConfig with NF4 quantization and bfloat16 compute dtype,␣
 ↪optimized for large models.
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
   # bnb_4bit_compute_dtype=torch.bfloat32
)

# Load the tokenizer for the large model.
tokenizer = AutoTokenizer.from_pretrained(model_id)

# Load the large model with the specified 4-bit quantization configuration.
# `device_map="auto"` is crucial here to distribute the model layers across␣
 ↪available devices.
model_4bit = AutoModelForCausalLM.from_pretrained(model_id,␣
 ↪quantization_config=bnb_config, device_map="auto")
```

```
Loading checkpoint shards:   0%|          | 0/46 [00:00<?, ?it/s]
```

[20]:
```
import re
import json
```

```
def get_response_json(text: str, user_query: str) -> str:
    """
    Return the JSON string that immediately follows the example:
        - User: "<user_query>"
      and the 'Response:' line.
    """
    # Look for "Strictly follow this format" and find the first JSON object␣
 ↪after that
    strict_pattern = r"Strictly follow this format for every query, and never␣
 ↪deviate\."
    strict_match = re.search(strict_pattern, text)

    if not strict_match:
        raise ValueError("Could not find 'Strictly follow this format' marker")

    # Search for JSON object starting from after the strict pattern
    search_text = text[strict_match.end():]

    # JSON object pattern that handles nested braces and quoted strings
    json_pattern = r'\{(?:[^{}"]|"[^"\\]*(?:\\.[^"\\]*)*")*\}'

    json_match = re.search(json_pattern, search_text)
    if not json_match:
        raise ValueError(f'No JSON object found after "Strictly follow this␣
 ↪format" marker')

    return json_match.group(0)

def get_response_json_as_dict(text: str, user_query: str) -> dict:
    """Same as above, but parsed into a Python dict."""
    return json.loads(get_response_json(text, user_query))

def route_query_quantized(user_query: str):
    router_system_prompt =f"""
    As a professional query router, your objective is to correctly classify␣
 ↪user input into one of three categories based on the source most relevant␣
 ↪for answering the query:
    1. "OPENAI_QUERY": If the user's query appears to be answerable using␣
 ↪information from OpenAI's official documentation, tools, models, APIs, or␣
 ↪services (e.g., GPT, ChatGPT, embeddings, moderation API, usage guidelines).
    2. "10K_DOCUMENT_QUERY": If the user's query pertains to a collection of␣
 ↪documents from the 10k annual reports, datasets, or other structured␣
 ↪documents, typically for research, analysis, or financial content.
```

```
   3. "INTERNET_QUERY": If the query is neither related to OpenAI nor the 10k␣
↪documents specifically, or if the information might require a broader search␣
↪(e.g., news, trends, tools outside these platforms), route it here.

   Your decision should be made by assessing the domain of the query.

   Always respond in this valid JSON format:
   {{
       "action": "OPENAI_QUERY" or "10K_DOCUMENT_QUERY" or "INTERNET_QUERY",
       "reason": "brief justification",
       "answer": "AT MAX 5 words answer. Leave empty if INTERNET_QUERY"
   }}

   EXAMPLES:

   - User: "How to fine-tune GPT-3?"
   Response:
   {{
       "action": "OPENAI_QUERY",
       "reason": "Fine-tuning is OpenAI-specific",
       "answer": "Use fine-tuning API"
   }}

   - User: "Where can I find the latest financial reports for the last 10␣
↪years?"
   Response:
   {{
       "action": "10K_DOCUMENT_QUERY",
       "reason": "Query related to annual reports",
       "answer": "Access through document database"
   }}

   - User: "Top leadership styles in 2024"
   Response:
   {{
       "action": "INTERNET_QUERY",
       "reason": "Needs current leadership trends",
       "answer": ""
   }}

   - User: "What's the difference between ChatGPT and Claude?"
   Response:
   {{
       "action": "INTERNET_QUERY",
       "reason": "Cross-comparison of different providers",
       "answer": ""
   }}
```

```
    Strictly follow this format for every query, and never deviate.
    User: "{user_query}"
    """

    try:
        # Query the GPT-4 model with the router prompt and user input

        device = "cuda:0"

        # Encode the input text using the tokenizer and move the resulting␣
↪tensors to the specified device.
        inputs = tokenizer(router_system_prompt, return_tensors="pt").to(device)

        # Generate a sequence of tokens from the model based on the input.
        # `max_new_tokens` limits the length of the generated text.


        outputs = model_4bit.generate(**inputs, max_new_tokens=200)

        # Decode the generated token IDs back into human-readable text and␣
↪print the output.

        task_response = tokenizer.decode(outputs[0], skip_special_tokens=True)
        return get_response_json_as_dict(task_response,  user_query)


    # Handle case where model response isn't valid JSON
    except json.JSONDecodeError as json_err:
        return {
            "action": "INTERNET_QUERY",
            "reason": f"JSON parsing error: {json_err}",
            "answer": ""
        }

    # Catch-all for any other unforeseen issues
    except Exception as err:
        return {
            "action": "INTERNET_QUERY",
            "reason": f"Unexpected error: {err}",
            "answer": ""
        }

route_query_quantized("what is the revenue of uber in 2021?")
```

```
Setting `pad_token_id` to `eos_token_id`:0 for open-end generation.
```

[20]:
```
{'action': '10K_DOCUMENT_QUERY',
 'reason': 'Query related to annual reports',
 'answer': 'Access through document database'}
```

The router using the quantized model seems to be working. So let's upda the agentic_rag below to see if it matches to openai's

3.0.1 Note

After some experimentation it is difficult to use a quantized LLM at least for the ones I have tried to get good generated responses in RAG. Therefore we are using for this notebook to use quantized models just for the routing. We will continue to use OpenAI for the RAG responses.

[ ]:

[21]:
```
def agentic_rag_quantized(user_query: str):
    """
    Main function that runs the full Agentic RAG system.

    This function takes a user's question, decides what type of query it is␣
↪(OpenAI-related,
    financial document-related, or general internet), and then calls the right␣
↪function
    to handle it. Finally, it prints out the full conversation and response.

    Args:
        user_query (str): The user's input question.

    Returns:
        None (It just prints the result nicely to the console)
    """

    #  Terminal color codes to make the printed output easier to read and␣
↪visually structured
    CYAN = "\033[96m"
    GREY = "\033[90m"
    BOLD = "\033[1m"
    RESET = "\033[0m"

    try:
        # Step 1: Print the user's original question to the console
        print(f"{BOLD}{CYAN} User Query:{RESET} {user_query}\n")

        # Step 2: Use the router (powered by GPT) to decide which route the␣
↪query belongs to
        try:
            response = route_query_quantized(user_query)
        except Exception as route_err:
```

```
            # If something goes wrong while classifying the query, show an␣
↪error message
            print(f"{BOLD}{CYAN}  BOT RESPONSE:{RESET}\n")
            print(f"Routing error: {route_err}\n")
            return

        # Extract the routing decision and the reason behind it
        action = response.get("action")  # e.g., "OPENAI_QUERY"
        reason = response.get("reason")  # e.g., "Related to OpenAI tools"

        # Step 3: Show the selected route and why it was chosen
        print(f"{GREY}  Selected Route: {action}")
        print(f"  Reason: {reason}")
        print(f"  Processing query...{RESET}\n")

        # Step 4: Call the correct function depending on the route (retrieval␣
↪or web search)
        try:
            route_function = routes.get(action)  # Find the function to use for␣
↪this route
            if route_function:
                result = route_function(user_query, action)  # Run the function␣
↪with the user's input
            else:
                result = f"Unsupported action: {action}"  # Catch unknown␣
↪routing types
        except Exception as exec_err:
            result = f"Execution error: {exec_err}"  # Handle failure in the␣
↪chosen route function

        # Step 5: Print the final response to the user
        print(f"{BOLD}{CYAN}  BOT RESPONSE:{RESET}\n")
        print(f"{result}\n")

    except Exception as err:
        # Catch-all for any unexpected errors in the overall logic
        print(f"{BOLD}{CYAN}  BOT RESPONSE:{RESET}\n")
        print(f"Unexpected error occurred: {err}\n")
```

[22]:
```
agentic_rag("what was uber revenue in 2021?")
```

```
User Query: what was uber revenue in 2021?

Selected Route: 10K_DOCUMENT_QUERY

Reason: Query related to historical financial data

Processing query…
```

```
 BOT RESPONSE:

The revenue of Uber in 2021 was $17,455 million [1].
```

[23]:
```
agentic_rag("what was lyft revenue in 2024?")
```

```
 User Query: what was lyft revenue in 2024?

 Selected Route: INTERNET_QUERY

 Reason: Future revenue requires external data

 Processing query…

Getting your response from the internet  …
 BOT RESPONSE:

Lyft revenue in 2024 was $5.8 billion, a 31% increase from 2023.
```

[24]:
```
agentic_rag("List me down new LLMs in 2025")
```

```
 User Query: List me down new LLMs in 2025

 Selected Route: INTERNET_QUERY

 Reason: Future state of LLMs in 2025

 Processing query…

Getting your response from the internet  …
 BOT RESPONSE:

New Large Language Models in 2025:

1. GPT-4.5 (Orion) - Released on February 27, 2025, by OpenAI.
2. Claude 3.7 Sonnet - Released on February 24, 2025, by Anthropic.
3. Gemini 2.5 Pro - Released in February 2025, by Google DeepMind.
4. DeepSeek-V3-0324 - Released in 2025, by an unknown developer.
5. Grok-3 - Released in 2025, by an unknown developer.
6. Qwen3 - Released in 2025, by an unknown developer.
7. Llama 4 - Released in 2025, by Meta.
8. Claude Sonnet 4 - Released in May 2025, by Anthropic.
9. LLaMA 3 - Released in 2025, by Meta.
10. MiniMax-Text-01 - Released in January 2025, by Minimax.
11. Gemini 2.0 - Released in February 2025, by Google DeepMind.

These are the new large language models in 2025, according to the provided
```

sources.

[25]:
```
agentic_rag("how to work with chat completions?")
```

```
User Query: how to work with chat completions?

Selected Route: OPENAI_QUERY

Reason: Chat completions relate to OpenAI

Processing query…

BOT RESPONSE:
```

To work with chat completions, you have access to various endpoints in OpenAI's API [1]. Given a unique ID, you can retrieve specific chat completion messages [1]. This is done by sending a GET request to `https://api.openai.com/v1/chat/completions/{completion_id}/messages` where `{completion_id}` is replaced by the unique ID of the chat completion [1].

To retrieve this, you would use the headers for authorization (with your API key) and content type set to `application/json` [1]. The response would be a list of messages containing IDs, roles (user/assistant), and content [1].

To list stored chat completions, you can send a GET request to `https://api.openai.com/v1/chat/completions` [1]. Only chat completions that have the `store` parameter set to `true` will be returned [1].

You can also time-specifically stream chat completions in real-time and receive chunks of completions returned from the model using server-sent events [1]. Object type for these chat completions would be `chat.completion.chunk` [1][2].

You can set query parameters such as `after` to get the last message from the previous pagination request, `limit` to set the number of messages to retrieve, and `order` to sort messages by timestamp in `asc` or `desc` order [1].

The processed request would also provide metadata such as `total_tokens`, `prompt_tokens_details`, and `completion_tokens_details` [2]. Remember to check the `has_more` field in the response to see if there are more pages of messages to retrieve [1].