Tutorial: Corrective RAG

Large language models (LLMs) are incredibly powerful, but their knowledge is limited to their training datasets. When answering questions, especially about specific, evolving or proprietary information, LLMs can hallucinate or provide general, irrelevant answers. Retrieval augmented generation (RAG) helps by giving the LLM relevant retrieved information from external data sources.

However, not all RAG is created equal. Corrective retrieval augmented generation (cRAG) does not simply build on top of the more traditional RAG, it represents a significant improvement. It's devised to be more robust by evaluating the quality and relevance of the retrieved results. If the context is weak, irrelevant or from an untrustworthy source, cRAG attempts to find better information through corrective actions or explicitly refuse to answer rather than fabricating a response. This technique makes cRAG systems more reliable and trustworthy for critical applications like answering policy-related questions.

In this tutorial, you'll learn how to build a robust corrective RAG (cRAG) system by using IBM® Granite® models on Watsonx® and LangChain. Similar frameworks such as LlamaIndex or LangGraph can also be used for building complex RAG flows with distinct nodes. Techniques like fine-tuning can further enhance specific LLM performance for domain-specific RAG. LLMs like those from OpenAI (for example, GPT models like ChatGPT) are also popular choices for such agents, though this tutorial focuses on IBM Granite.

Here, we'll focus on a use case: answering questions about a specific insurance policy document (a PDF). This tutorial will guide you in implementing a sophisticated RAG algorithm that:

- Retrieves information from your own PDF document.
- If the internal documents are not sufficient for generating the answer, the agent can use an external web search (Tavily) as a fallback.
- The agent intelligently filters out irrelevant external results so the answers are tailored to private policies.
- The agent will give clear, limited responses with partial information when available or a clear refusal where context is missing.

Use case: Building a reliable insurance policy query agent

This tutorial is a demonstration of creating an insurance policy query agent designed to analyze policy documents (a PDF brochure) and answer user queries accurately. We use IBM Granite models and LangChain to build the agent with robust retrieval and verification steps ensuring high-quality, source-constrained answers.

Let's understand how the key principles of reliable RAG apply in our use case.

Application of key principles

Internal knowledge base (PDF): The agent's primary source of truth is your provided insurance policy PDF. It converts this document into a searchable vector store.

External search fallback (Tavily): If the internal knowledge base doesn't have enough information, the agent can consult external web sources through Tavily. Tavily is a search engine built specifically for AI agents and LLMs that results in faster and real-time retrieval through its application programming interface (API) for RAG-based applications.

Context scoring: The LLM-based retrieval evaluator (acting as a grader) will provide a score to the relevance of the items retrieved from your internal PDF while ensuring that only high-quality retrieved items are included.

Query rewriting: For web searches, the agent can rephrase the user's query to improve the chances of finding relevant external information.

Source verification: An LLM-powered check evaluates whether external web search results are relevant to a private insurance policy, filtering out general information or details about public health programs (like Medi-Cal). This function prevents the generation of misleading answers and enables self-correction, aiding in knowledge refinement.

Constrained generation: The final prompt to the LLM strictly instructs it to use only the provided context, offer exact answers, state when information is unavailable or provide partial answers with explicit limitations. This function enhances the adaptability and reliability of the generated responses.

Prerequisites

You need an IBM Cloud® account to create a watsonx.ai® project. Ensure that you have access to both your watsonx API Key and Project ID. You will also need an API key for Tavily AI for web search capabilities.

Steps

Step 1. Set up your environment

While you can choose from several tools, this tutorial walks you through how to set up an IBM account by using a Jupyter Notebook.

1. Log in to watsonx.ai by using your IBM Cloud account.
2. Create a watsonx.ai project. You can get your project ID from within your project. Click the Manage tab. Then, copy the project ID from the Details section of the General page. You need this ID for this tutorial.
3. Create a Jupyter Notebook.

This step opens a notebook environment where you can copy the code from this tutorial. Alternatively, you can download this notebook to your local system and upload it to your watsonx.ai project as an asset. To view more Granite tutorials, check out the IBM Granite Community. This tutorial is also available on GitHub.

Step 2. Set up watsonx.ai runtime service and API key

1. Create a watsonx.ai Runtime service instance (choose the Lite plan, which is a free instance).
2. Generate an application programming interface (API) Key.
3. Associate the watsonx.ai Runtime service to the project that you created in watsonx.ai.

Step 3. Installation of the packages

To work with the LangChain framework and integrate IBM WatsonxLLM, we need to install some essential libraries. Let's start by installing the required packages. This set includes **langchain** for the RAG framework, **langchain-ibm** for the watsonx integration, **faiss-cpu** for efficient vector storage, **PyPDF2** for processing PDFs, **sentence-transformers** for getting an embedding and **requests** for web API calls. These libraries are critical to applying machine learning and NLP solutions.
Install Libraries !pip install langchain langchain-ibm faiss-cpu PyPDF2 sentence-transformers requests

Note: No GPU is required, but execution can be slower on CPU-based systems. This step opens a notebook environment where you can copy the code from this tutorial. This tutorial is also available on GitHub.

Step 4. Import required libraries

Next, import all the required modules and securely provide your API keys for watsonx and Tavily, along with your watsonx Project ID.
Import required libraries import os import io import getpass from PyPDF2 import PdfReader from langchain_ibm import WatsonxLLM from langchain.embeddings import HuggingFaceEmbeddings from langchain.vectorstores import FAISS from langchain.text_splitter import RecursiveCharacterTextSplitter from langchain.schema import Document import requests from botocore.client import Config import ibm_boto3 from langchain.prompts import PromptTemplate from langchain.tools import BaseTool # Watsonx WML_URL = "https://us-south.ml.cloud.ibm.com" WML_API_KEY = getpass.getpass(" Enter Watsonx API Key: ") PROJECT_ID = input(" Enter Watsonx Project ID: ") # Tavily TAVILY_API_KEY = getpass.getpass(" Enter Tavily API Key: ") print(" Credentials loaded.")

os helps to work with the operating system.

io allows for working with streams of data.

getpass uses a safe way to capture sensitive information like API keys and doesn't display input to the screen.

PyPDF2.PdfReader allows for content extraction from PDFs.

langchain_ibm.WatsonxLLM allows us to use the IBM Watsonx Granite LLM easily within the LangChain framework.

langchain.embeddings.HuggingFaceEmbeddings takes a HuggingFace model and generates the textual embeddings that are important for semantic search.

langchain.vectorstores.FAISS is a library for efficient vector storage and similarity search that allows us to build a vector index and query it.

langchain.text_splitter.RecursiveCharacterTextSplitter helps split large constituents of text into the smaller chunks needed to process documents that would not otherwise fit into memory.

langchain.schema.Document represents an arbitrary unit of text with associated metadata making it a building block in langchain.

requests is used for making HTTP requests externally to APIs.

botocore.client.Config is a configuration class used to define configuration settings for an AWS/IBM Cloud Object Storage client.

ibm_boto3 is the IBM Cloud Object Storage SDK for Python that helps to interact with cloud object storage.

langchain.prompts.PromptTemplate offers a way to create reusable, structured prompts for language models.

langchain.tools.BaseTool is the base class from which you build custom tools that can be given to LangChain agents for use.

This step sets up all the tools and modules that we need to process text, create embeddings, store them in a vector database and interact with the IBM watsonx LLM. It establishes all the parts needed to create a real-world RAG system, capable of sourcing, querying and searching a range of data types.

Step 5. Load and process a PDF from IBM Cloud Object Storage

In this step, we will load the insurance policy PDF from IBM Cloud Object Storage. The code reads the PDF, reads the text content and splits the text into smaller and manageable chunks. These chunks are converted into numerical embeddings and stored in a FAISS vector store that prepares us for semantic similarity search later in local context to optimize retrieval results.
import os, types import pandas as pd from botocore.client import Config import ibm_boto3 def __iter__(self): return 0 cos_client = ibm_boto3.client(service_name='s3', ibm_api_key_id='YOUR_IBM_API_KEY', ibm_auth_endpoint="https://iam.cloud.ibm.com/identity/token", config=Config(signature_version='oauth'), endpoint_url='https://s3.direct.us-south.cloud-object-storage.appdomain.cloud') bucket = 'YOUR_BUCKET_NAME' object_key = 'YOUR_OBJECT_KEY' streaming_body_2 = cos_client.get_object(Bucket=bucket, Key=object_key)['Body'] pdf_bytes = io.BytesIO(streaming_body_2.read()) reader = PdfReader(pdf_bytes) text = "" for page in reader.pages: extracted = page.extract_text() if extracted: text += extracted print(f" Extracted {len(text)} characters from PDF.")
splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50) chunks = splitter.split_text(text) print(f" Split into {len(chunks)} chunks.") embeddings = HuggingFaceEmbeddings(model_name="sentence-transformers/all-MiniLM-L6-v2") vectorstore = FAISS.from_texts(chunks, embeddings) print(f" Created FAISS index.")

ibm_boto3.client enables the client to interact with IBM Cloud Object Storage.

Bucket is the name of the cloud object storage bucket that contains the PDF.

object_key is the name of the PDF in the cloud object storage bucket.

cos_client.get_object(...).read() retrieves the content of the PDF file in cloud object storage as bytes.

io.BytesIO converts the PDF raw bytes into an in-memory binary stream in a format that can be used by PdfReader.

PdfReader creates an object that can parse and extract text from the PDF.

page.extract_text() extracts the text of a single page in the PDF.

RecursiveCharacterTextSplitter is configured to split the extracted text into chunks of 500 characters with an overlap of 50 characters, therefore keeping everything in context.

splitter.split_text(text) runs the splitting of all pages of the PDF text into the smaller chunks.

HuggingFaceEmbeddings loads a sentence transformer model that has been pretrained to convert the text chunks into dense vector representations.

FAISS.from_texts(chunks, embeddings) builds an in-memory FAISS index that enables chunks of text to be searchable by their semantic similarities.

This step handles full ingestion of a PDF document from cloud to LLM-ready text and comfortable indexing for real-time retrieval.

Step 6. Initialize LLM and tools

In this step, you'll configure the IBM Granite LLM to drive your agent's reasoning and integrate it with the Tavily web search function. The parameters of the LLM are set up for factual, stable responses.
llm = WatsonxLLM(model_id="ibm/granite-3-2b-instruct", url=WML_URL, apikey=WML_API_KEY, project_id=PROJECT_ID, params={ "max_new_tokens": 300, # ~2-3 paragraphs, good for corrective RAG "temperature": 0.2, # low temperature = more factual, stable answers }) print(" Watsonx Granite LLM ready.")
class TavilySearch(BaseTool): name: str = "tavily_search" description: str = "Search the web using Tavily for extra info." def _run(self, query: str): response = requests.post("https://api.tavily.com/search", json={"api_key": TAVILY_API_KEY, "query": query}) response.raise_for_status() return response.json()['results'][0]['content'] tavily_tool = TavilySearch()

WatsonxLLM instantiates the LLM wrapper for IBM watsonx, allowing interaction with Granite models.

model_id="ibm/granite-3-2b-instruct" is the IBM Granite model (a 2.7 billion parameter instruct model) designed for instruction-based generative AI tasks.

class TavilySearch(BaseTool) defines a custom LangChain tool for performing web searches by using the Tavily API.

tavily_tool = TavilySearch() creates an executable instance of the custom Tavily search tool.

When we initialize watsonxLLM, the **url**, **apikey** and **project_id** values from our previously set up credentials are passed to authenticate and connect to the service. Its parameters, such as **"max_new_tokens": 300**, limit the response length and **"temperature": 0.2** controls output creativity, favoring more deterministic results.

The **TavilySearch** class definition includes a description of its function. Its logic is contained within the **def _run(self, query: str)** method. In this method, we make an HTTP POST request to the Tavily API endpoint, including the **TAVILY_API_KEY** and the search query in the JSON payload. We then

verify if there are any HTTP errors with **response.raise_for_status()** and parse the JSON response to access the content snippet from the first search result.

This step sets up the language model for text generation and includes an external web search tool as a way to augment the language model knowledge.

Step 7. Define prompt templates and helper functions

This step defines the various prompt templates that guide the LLM's behavior at different stages of the RAG process. This approach includes prompts for scoring the relevance of internal document chunks, rewriting user queries for better web search and a critical new prompt for verifying the source of web search results. Helper functions for scoring chunks and retrieving them from the vector store are also defined.

```
# Define Prompt Templates and Helper Functions # Prompt for scoring the relevance of retrieved chunks scoring_prompt_template = PromptTemplate.from_template( """ You are an evaluator. Score the relevance of the context chunk to the given insurance question. Question: "{query}" Context: \"\"\" {chunk} \"\"\" Respond only in this format: Score: <0-5> Reason: <one line reason> """ ) # Prompt for rewriting the user's query for better web search results rewrite_prompt_template = PromptTemplate.from_template( """ You are a helpful assistant. Improve the following question to be clearer for an insurance information search. Focus on making the query more specific if possible. Original Question: "{query}" Rewrite it to be clearer: """ ) # NEW: Prompt for verifying if Tavily context is from a relevant source (private policy vs. public program) CONTEXT_SOURCE_VERIFICATION_PROMPT = PromptTemplate.from_template( """ You are an expert at identifying if a piece of text is from a general, public, or unrelated source versus a specific, private, or relevant policy document. Read the following context and determine if it appears to discuss general information, public health programs (like Medi-Cal, Medicaid, Medicare, NHS, government-funded programs, state-funded), or information that is clearly *not* specific to a private insurance policy like the one the user might be asking about (assuming the user is asking about their own private policy). If the context explicitly mentions or heavily implies public health programs, or is too general to be useful for a specific private policy question, respond with "NO". Otherwise (if it seems like it *could* be from a private policy context, a general insurance guide, or does not explicitly mention public programs), respond with "YES". Context: \"\"\" Response: """ ) # Function to score chunks using the LLM def score_chunks(chunks, query): scored = [] for chunk in chunks: prompt = scoring_prompt_template.format(query=query, chunk=chunk) response = llm(prompt).strip() try: # Extract score using more robust parsing score_line = [line for line in response.splitlines() if "Score:" in line] if score_line: score = int(score_line[0].replace("Score:", "").strip()) else: score = 0 # Default to 0 if score line not found except Exception as e: print(f" Could not parse score for chunk: {e}. Response: {response[:50]}...") score = 0 # Default to 0 on error scored.append((chunk, score)) return scored # Function to retrieve documents from FAISS vector store def retrieve_from_vectorstore(query): # Retrieve top 8 similar documents from your PDF content docs = vectorstore.similarity_search(query, k=8) return [doc.page_content for doc in docs] print(" Prompt templates and helper functions defined.")
```

This step defines the various prompt templates that guide the LLM's behavior at different stages of the RAG process. Prompts for scoring the relevance of internal document chunks, rewriting user queries for better web search and a critical new prompt for verifying that the source of web search results are included. Helper functions for scoring chunks and retrieving them from the vector store are also defined.

PromptTemplate.from_template is a utility function from LangChain to create a reusable template for constructing prompts.

scoring_prompt_template defines a prompt that instructs the LLM to act as an evaluator and assign a relevance score (0–5) to a specific context chunk based on a question.

rewrite_prompt_template defines a prompt that guides the LLM to improve or make a user's original question clearer for searching.

CONTEXT_SOURCE_VERIFICATION_PROMPT defines a prompt that instructs the LLM to verify whether a piece of text (for example, from web search) is from a private policy context or a general or public source.

def score_chunks(chunks, query) defines a function that takes a list of text chunks and a query then uses the LLM to score the relevance of each chunk.

def retrieve_from_vectorstore(query) defines a function to retrieve the most similar documents from the FAISS vector store.

Within the **score_chunks** function, an empty scored list is initialized. For each chunk, the **scoring_prompt_template** is formatted with the specific query and chunk. This formatted prompt is then sent to the LLM and the response is stripped. The function attempts to extract the integer score (a binary score if simplified to relevant or not relevant) by identifying the "Score:" line in the model's response. The chunk along with its parsed or defaulted score is then added to the scored list. This part of the system acts as a retrieval evaluator or grader.

The function **retrieve_from_vectorstore** implements a **vectorstore.similarity_search** to find the 8 most relevant document chunks based on the query and retrieves the **page_content** from these retrieved LangChain Document objects.

This step builds the conceptual scaffolding for the corrective RAG system so that both the LLM will evaluate context and how to retrieve knowledge from both internal and external knowledge sources.

Step 8. Implement the corrective RAG logic

Initial retrieval is the function that scans the vector store of the PDF.

Context scoring takes the PDF chunks that have been retrieved to context score according to relevancy.

Fallback to tavily if there's not enough relevant context from the PDF, it then queries Tavily (web search).

Source verification is an LLM-powered step that checks if the Tavily results are relevant to a private policy before using them. This function prevents misleading answers from public health programs.

Query rewriting and second tavily search if there's still not any good context, it rewrites the query and tries Tavily search again.

Final decision when there is any relevant context, it is sent to the LLM with a (strict) prompt to create the answer. If there is no relevant context after all viable attempts, it sends a polite denial.
Implement the Corrective RAG Logic MIN_CONTEXT_LENGTH = 100 # Adjust this based on how much minimal context you expect for a partial answer SIMILARITY_THRESHOLD = 3 # Only scores >= 3 used for vector store chunks def corrective_rag(query: str, policy_context_keywords: list = None): """ Executes the Corrective RAG process to answer insurance queries. Args: query (str): The user's question. policy_context_keywords (list, optional): Keywords related to the specific policy (e.g.,

["Super Star Health", "Care Health Insurance"]). Used to make external searches more specific. Defaults to None. Returns: str: The final answer generated by the LLM or a predefined refusal. """ retrieved_context_pieces = [] # To store all relevant pieces found throughout the process # Initial vector search & Scoring (from your PDF) chunks_from_vectorstore = retrieve_from_vectorstore(query) scored_chunks_vector = score_chunks(chunks_from_vectorstore, query) good_chunks_vector = [chunk for chunk, score in scored_chunks_vector if score >= SIMILARITY_THRESHOLD] retrieved_context_pieces.extend(good_chunks_vector) current_context = "\n\n".join(retrieved_context_pieces) print(f" Context length after initial vector scoring: {len(current_context)}") # Prepare specific query for Tavily by optionally adding policy keywords tavily_search_query = query if policy_context_keywords: tavily_search_query = f"{query} {' '.join(policy_context_keywords)}" # Fallback: Tavily direct search (only if current context is too short from vector store) if len(current_context) < MIN_CONTEXT_LENGTH: print(f" Context too short from internal docs, trying Tavily direct with query: '{tavily_search_query}'...") tavily_context_direct = tavily_tool._run(tavily_search_query) if tavily_context_direct: # --- NEW STEP: Verify Tavily Context Source --- # Ask the LLM if the Tavily result seems to be from a private policy context or a public program verification_prompt = CONTEXT_SOURCE_VERIFICATION_PROMPT.format(context=tavily_context_direct) is_relevant_source = llm(verification_prompt).strip().upper() if is_relevant_source == "YES": retrieved_context_pieces.append(tavily_context_direct) current_context = "\n\n".join(retrieved_context_pieces) # Re-combine all good context print(f" Context length after Tavily direct (verified and added): {len(current_context)}") else: print(f" Tavily direct context source rejected (e.g., public program): {tavily_context_direct[:100]}...") # Context is NOT added, so it remains short and triggers the next fallback or final refusal # Fallback: Rewrite query + Tavily (only if context is still too short after direct Tavily) if len(current_context) < MIN_CONTEXT_LENGTH: print(" Context still too short, rewriting query and trying Tavily...") rewrite_prompt = rewrite_prompt_template.format(query=query) improved_query = llm(rewrite_prompt).strip() # Add policy keywords to the rewritten query too if policy_context_keywords: improved_query = f"{improved_query} {' '.join(policy_context_keywords)}" print(f" Rewritten query: '{improved_query}'") tavily_context_rewritten = tavily_tool._run(improved_query) if tavily_context_rewritten: # --- NEW STEP: Verify Rewritten Tavily Context Source --- verification_prompt = CONTEXT_SOURCE_VERIFICATION_PROMPT.format(context=tavily_context_rewritten) is_relevant_source = llm(verification_prompt).strip().upper() if is_relevant_source == "YES": retrieved_context_pieces.append(tavily_context_rewritten) current_context = "\n\n".join(retrieved_context_pieces) # Re-combine all good context print(f" Context length after rewritten Tavily (verified and added): {len(current_context)}") else: print(f" Tavily rewritten context source rejected (e.g., public program): {tavily_context_rewritten[:100]}...") # --- Final Decision Point --- # Now, `current_context` holds ALL the "good" and "verified" context we managed to gather. # The decision to call the LLM for an answer or give a hard refusal is based on `current_context`'s length. # Final check for absolutely no good context # This triggers only if *no* relevant internal or external context was found or verified. if len(current_context.strip()) == 0: print(" No good context found after all attempts. Returning absolute fallback.") return ("Based on the information provided, there is no clear mention of this specific detail " "in the policy documents available.") # If we have *any* context (even if short), pass it to the LLM to process # The LLM will then decide how to phrase the answer based on its prompt instructions # (exact, partial, or full refusal if context is irrelevant or insufficient based on its own reasoning). final_prompt = (f"You are a careful insurance expert.\n" f"Use ONLY the following context to answer the user's question. If the context is too short " f"or does not contain the answer, you must indicate that.\n" f"Context:\n```\n{current_context}\n```\n\n" # Pass the gathered context f"User's Question: {query}\n\n" # Pass the original query for the LLM's reference f"NEVER add new details that are not in the context word-for-word.\n" f"If the context clearly says the answer, give it exactly as written in the context, but in prose.\n" f"If the context does not mention the topic at all, or the answer is not in the context, say:\n" f"\"I'm sorry, but this information is not available in the provided policy details.\"\n" f"If the context partially mentions the topic but does not directly answer the specific question (e.g., mentions 'dental' but not 'wisdom tooth removal'), reply like this:\n"

f"\"Based on the information provided, here's what is known: [quote relevant details from the context related to the broad topic.] " f"There is no clear mention of the specific detail asked about.\"\n" f"Do NOT assume. Do NOT make up extra information.\n" f"Do NOT generate extra questions or conversational filler.\n" f"Final Answer:") return llm(final_prompt) print(" Corrective RAG logic implemented.")

The first pass of the **policy_context_keywords** parameter allows you to add specific terms from your policy (for example, its name, insurer) to help narrow searches for Tavily.

MIN_CONTEXT_LENGTH defines the minimum acceptable length of retrieved context.

SIMILARITY_THRESHOLD defines the minimum relevance score that a chunk must have to be considered "good."

def corrective_rag(...) defines the main function that orchestrates the entire corrective RAG workflow.

The **corrective_rag** function begins by creating **retrieved_context_pieces** to gather relevant context. It first fetches and scores **chunks_from_vectorstore** from the PDF vector store based on the query, then **scored_chunks_vector** evaluates their relevance by using the language model.
Only **good_chunks_vector** that meet the **SIMILARITY_THRESHOLD** are kept.
The **current_context** is then compiled from these pieces.

If the **current_context** is below **MIN_CONTEXT_LENGTH**, the system attempts a web search. It constructs **tavily_search_query**, potentially incorporating **policy_context_keywords**. A direct search (**tavily_context_direct**) is performed. Crucially, a verification_prompt is created and sent to the LLM to determine whether the web search result (**is_relevant_source**) is from a private policy rather than a public program. If it's YES, the context is added.

If the context remains insufficient, the system prepares to rewrite the query. It uses **rewrite_prompt** to get an **improved_query** from the LLM, then performs a second web search (**tavily_context_rewritten**). This new context also undergoes the same source verification.

Finally, **if len(current_context.strip()) == 0** is a last check. If no relevant context is found after all attempts, a predefined refusal message is returned. Otherwise, a **final_prompt** is created with all the verified context and sent to the language model to generate its final answer.

The entire **corrective_rag** function handles the staged retrieving, scoring and verifying functions of corrective RAG in detail. It allows for constant updating of the knowledge base and knowledge stream and brings the benefit of robust and contextually aware answers.

Step 9. Test the system

Finally, execute the **corrective_rag** function with a sample query. It's crucial to provide **policy_context_keywords** that are specific to your PDF document. These keywords will help the Tavily web search become more relevant to your actual policy, preventing general or public health program information from polluting your context.

Observe the print statements for context length and verification results to understand the flow of information.
query = "How does the policy cover for In-Patient Hospitalisation?" result = corrective_rag(query) print("\n FINAL ANSWER:\n") print(result)

policy_specific_keywords = ["Super Star Health", "Care Health Insurance"] defines a list of keywords that are relevant to the uploaded insurance policy, helping to narrow down web search results.

query = "..." defines the particular question that a user might ask.

result = corrective_rag(query, policy_context_keywords=policy_specific_keywords) calls the main **corrective_rag** function and passes the user's query and policy-specific keywords to begin the entire RAG process.

print("\n FINAL ANSWER (...)") displays a clear header before printing the generated answer.

print(result) outputs the final answer returned by the **corrective_rag** system.

This step shows how to invoke the complete corrective RAG system with a sample query and keywords, demonstrating its end-to-end functionality in a real-world scenario.

Key takeaways

The corrective RAG implemented fully coordinated an internal PDF knowledge base with external service (Tavily) to retrieve comprehensive information for complex requests.

It accurately evaluated and filtered through retrieved context by using LLM-based scoring and critical source verification to ensure valid and reliable information is being used.

The system demonstrated the ability to improve external search by intelligently rewriting user queries to request more targeted and higher-quality information.

By using constrained generation, a reliable and contextually accurate answer was commonly generated and the system politely refused to answer if there was not enough known verified information.

This example demonstrated how LangChain and IBM Granite LLMs on watsonx can be used to develop powerful and trustworthy AI-based applications in sensitive domains such as asking questions about insurance policies.