Build a LangChain agentic RAG system using Granite in watsonx.ai

Introduction

In this tutorial, you will create a LangChain agentic RAG system using the IBM Granite-3.2-8B-Instruct model available on watsonx.ai that can answer complex queries about the 2024 US Open using external information.

Overview of agentic RAG

What is RAG?

Retrieval-augmented generation (RAG) is a technique in natural language processing (NLP) that leverages information retrieval and generative models to produce more accurate, relevant and contextually aware responses. In traditional language generation tasks, large language models (LLMs) such as Meta's Llama Models or IBM's Granite Models are used to construct responses based on an input prompt. Common real-world use cases of these large language models are chatbots. When models are missing relevant information that is up to date in their knowledge base, RAG is a powerful tool.

What are AI agents?

At the core of agentic RAG systems are artificial intelligence (AI) agents. An AI agent refers to a system or program that is capable of autonomously performing tasks on behalf of a user or another system by designing its workflow and using available tools. Agentic technology implements tool use on the backend to obtain up-to-date information from various data sources, optimize workflow and create subtasks autonomously to solve complex tasks. These external tools can include external data sets, search engines, APIs and even other agents. Step-by-step, the agent reassesses its plan of action in real time and self-corrects.

Agentic RAG versus traditional RAG

Agentic RAG frameworks are powerful as they can encompass more than just one tool. In traditional RAG applications, the LLM is provided with a vector database to reference when forming its responses. In contrast, agentic AI applications are not restricted to document agents that only perform data retrieval. RAG agents can also have tools for tasks such as solving mathematical calculations, writing emails, performing data analysis and more. These tools can be supplemental to the agent's decision-making process. AI agents are context-aware in their multistep reasoning and can determine when to use appropriate tools.

AI agents, or intelligent agents, can also work collaboratively in multiagent systems, which tend to outperform singular agents. This scalability and adaptability is what sets apart agentic RAG agents from traditional RAG pipelines.

Prerequisites

You need an IBM Cloud® account to create a watsonx.ai™ project.

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 to use a Jupyter Notebook.

1. Log in to watsonx.ai 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 will open 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 Jupyter Notebook along with the datasets used can be found on GitHub.

Step 2. Set up a watsonx.ai Runtime instance and API key

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

Step 3. Install and import relevant libraries and set up your credentials

We need a few dependencies for this tutorial. Make sure to import the following ones; if they're not installed, you can resolve this with a quick pip installation.

Common Python frameworks for building agentic AI systems include LangChain, LangGraph and LlamaIndex. In this tutorial, we will be using LangChain.
installations !pip install langchain | tail -n 1 !pip install langchain-ibm | tail -n 1 !pip install langchain-community | tail -n 1 !pip install ibm-watsonx-ai | tail -n 1 !pip install ibm_watson_machine_learning | tail -n 1 !pip install chromadb | tail -n 1 !pip install tiktoken | tail -n 1 !pip install python-dotenv | tail -n 1 !pip install bs4 | tail -n 1
Powered by Granite

Now, import the following modules and classes.
imports import getpass import os from dotenv import load_dotenv from langchain_ibm import WatsonxEmbeddings, WatsonxLLM from langchain.vectorstores import Chroma from langchain_community.document_loaders import WebBaseLoader from langchain.text_splitter import RecursiveCharacterTextSplitter from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder from langchain.prompts import PromptTemplate from langchain.tools import tool from langchain.tools.render import render_text_description_and_args from langchain.agents.output_parsers import JSONAgentOutputParser from langchain.agents.format_scratchpad import format_log_to_str from langchain.agents import AgentExecutor from langchain.memory import ConversationBufferMemory from langchain_core.runnables import RunnablePassthrough from ibm_watson_machine_learning.metanames import GenTextParamsMetaNames as GenParams from ibm_watsonx_ai.foundation_models.utils.enums import EmbeddingTypes
Powered by Granite

Now, it's time to set up your credentials.

Option 1:

Input your watsonx.ai API key and project ID that you created in steps 1 and 2 upon running the following cell.
credentials = { "url": "https://us-south.ml.cloud.ibm.com", "apikey": getpass.getpass("Please enter your watsonx.ai Runtime API key (hit enter): ") } project_id = getpass.getpass("Please enter your project ID (hit enter): ")
Powered by Granite

Option 2:

Thegetpass() approach works well if you are running your Jupyter Notebook in watsonx.ai or locally in your preferred IDE. However, if you are running this notebook on an IDE such as Visual Studio Code, you can also store your WATSONX_APIKEY and PROJECT_ID in a.env file in the same level of your directory as this notebook, and use the following approach instead.
load_dotenv(os.getcwd()+"/.env", override=True) credentials = { "url": "https://us-south.ml.cloud.ibm.com", "apikey": os.getenv("WATSONX_APIKEY", ""), } project_id = os.getenv("PROJECT_ID", "")
Powered by Granite

For either approach, change the API endpoint URL if you are in a different region using the official documentation.

Step 4. Initialization a basic agent with no tools

This step is important as it will produce a clear example of an agent's behavior with and without external data sources. Let's start by setting our parameters.

The model parameters available can be found here. We experimented with various model parameters, including temperature, minimum and maximum new tokens and stop sequences. Learn more about model parameters and what they mean in the watsonx docs. It is important to set our stop_sequences here in order to limit agent hallucinations. This tells the agent to stop producing further output upon encountering particular substrings. In our case, we want the agent to end its response upon reaching an observation and to not hallucinate a human response. Hence, one of ourstop_sequences is"Human:" and another is"Observation" to halt once a final response is produced.

For this tutorial, we suggest using IBM's Granite-3.2-8B-Instruct model as the LLM to achieve similar results. You are free to use any AI model of your choice. The foundation models available through watsonx can be found here. The purpose of these models in LLM applications is to serve as the reasoning engine that decides which actions to take.
llm = WatsonxLLM(model_id="ibm/granite-3-2-8b-instruct", url=credentials.get("url"), apikey=credentials.get("apikey"), project_id=project_id, params={ GenParams.DECODING_METHOD: "greedy", GenParams.TEMPERATURE: 0, GenParams.MIN_NEW_TOKENS: 5, GenParams.MAX_NEW_TOKENS: 250, GenParams.STOP_SEQUENCES: ["Human:", "Observation"], },)
Powered by Granite

We'll set up a prompt template in case you want to ask multiple questions.
template = "Answer the {query} accurately. If you do not know the answer, simply say you do not know." prompt = PromptTemplate.from_template(template)
Powered by Granite

And now we can set up a chain with our prompt and our LLM. This allows the generative model to produce a response.
agent = prompt | llm
Powered by Granite

Let's test to see how our agent responds to a basic query.
agent.invoke({"query": 'What sport is played at the US Open?'})
Powered by Granite

Output: ' Do not try to make up an answer.\n\nThe sport played at the US Open is tennis.'

The agent successfully responded to the basic query with the correct answer. In the next step of this tutorial, we will be creating a RAG tool for the agent to access relevant information about IBM's involvement in the 2024 US Open. As we have covered, traditional LLMs cannot obtain current information on their own. Let's verify this.
agent.invoke({"query": 'Where was the 2024 US Open Tennis Championship?'})
Powered by Granite

Output: ‘ Do not make up an answer.\n\nThe 2024 US Open Tennis Championship has not been officially announced yet, so the location is not confirmed. Therefore, I do not know the answer to this question.’

Evidently, the LLM is unable to provide us with the relevant information. The training data used for this model contained information prior to the 2024 US Open and without the appropriate tools, the agent does not have access to this information.

Step 5. Establish the knowledge base and retriever

The first step in creating the knowledge base is listing the URLs we will be extracting content from. In this case, our data source will be collected from our online content summarizing IBM’s involvement in the 2024 US Open. The relevant URLs are established in the urls list.
urls = [‘https://www.ibm.com/case-studies/us-open’, ‘https://www.ibm.com/sports/usopen’, ‘https://newsroom.ibm.com/US-Open-AI-Tennis-Fan-Engagement’, ‘https://newsroom.ibm.com/2024-08-15-ibm-and-the-usta-serve-up-new-and-enhanced-generative-ai-features-for-2024-us-open-digital-platforms’]
Powered by Granite

Next, load the documents using LangChain WebBaseLoader for the URLs we listed. We’ll also print a sample document to see how it loaded.
docs = [WebBaseLoader(url).load() for url in urls] docs_list = [item for sublist in docs for item in sublist] docs_list[0]
Powered by Granite

Output:
```
Document(metadata={'source': 'https://www.ibm.com/case-studies/us-open', 'title': 'U.S. Open | IBM', 'description': 'To help the US Open
```

In order to split the data in these documents to chunks that can be processed by the LLM, we can use a text splitter such as RecursiveCharacterTextSplitter . This text splitter splits the content on the following characters:[“\n\n”, “\n”, “ “, “”] . This is done with the intention of keeping text in the same chunks, such as paragraphs, sentences and words together.

Once the text splitter is initiated, we can apply it to our docs_list .
text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(chunk_size=250, chunk_overlap=0) doc_splits = text_splitter.split_documents(docs_list)
Powered by Granite

The embedding model that we are using is an IBM Slate™ model through the watsonx.ai embeddings service. Let’s initialize it.
embeddings = WatsonxEmbeddings(model_id=EmbeddingTypes.IBM_SLATE_30M_ENG.value, url=credentials[“url”], apikey=credentials[“apikey”], project_id=project_id,)
Powered by Granite

In order to store our embedded documents, we will use Chroma DB, an open source vector store.
vectorstore = Chroma.from_documents(documents=doc_splits, collection_name=”agentic-rag-chroma”, embedding=embeddings,)
Powered by Granite

To access information in the vector store, we must set up a retriever.
retriever = vectorstore.as_retriever()
Powered by Granite

Step 6. Define the agent's RAG tool

Let’s define the get_IBM_US_Open_context() tool our agent will be using. This tool’s only parameter is the user query. The tool description is also noted to inform the agent of the use of the tool. This way, the agent knows when to call this tool. This tool can be used by the agentic RAG system for routing the user query to the vector store if it pertains to IBM’s involvement in the 2024 US Open.
@tool def get_IBM_US_Open_context(question: str): “””Get context about IBM’s involvement in the 2024 US Open Tennis Championship.””” context = retriever.invoke(question) return context tools = [get_IBM_US_Open_context]
Powered by Granite

Step 7. Establish the prompt template

Next, we will set up a new prompt template to ask multiple questions. This template is more complex. It is referred to as a structured chat prompt and can be used for creating agents that have multiple tools available. In our case, the tool we are using was defined in Step 6. The structured chat prompt will be made up of a system_prompt , a human_prompt and our RAG tool.

First, we will set up the system_prompt . This prompt instructs the agent to print its “thought process,” which involves the agent’s subtasks, the tools that were used and the final output. This gives us insight into the agent’s function calling. The prompt also instructs the agent to return its responses in JSON Blob format.
system_prompt = “””Respond to the human as helpfully and accurately as possible. You have access to the following tools: {tools} Use a json blob to specify a tool by providing an action key (tool name) and an action_input key (tool input). Valid “action” values: “Final Answer” or {tool_names} Provide only ONE action per $JSON_BLOB, as shown:” ``` {{ “action”: $TOOL_NAME, “action_input”: $INPUT }} ``` Follow this format: Question: input question to answer Thought: consider previous and subsequent steps Action: ``` $JSON_BLOB ``` Observation: action result ... (repeat Thought/Action/Observation N times) Thought: I know what to respond Action: ``` {{ “action”: “Final Answer”, “action_input”: “Final response to human” }} Begin! Reminder to ALWAYS respond with a valid json blob of a single action. Respond directly if appropriate. Format is Action:```$JSON_BLOB```then Observation”””
Powered by Granite

In the following code, we are establishing the human_prompt . This prompt tells the agent to display the user input followed by the intermediate steps taken by the agent as part of the agent_scratchpad .
human_prompt = “””{input} {agent_scratchpad} (reminder to always respond in a JSON blob)”””
Powered by Granite

Next, we establish the order of our newly defined prompts in the prompt template. We create this new template to feature the system_prompt followed by an optional list of messages collected in the agent’s memory, if any, and finally, the human_prompt which includes both the human input and agent_scratchpad .
prompt = ChatPromptTemplate.from_messages([(“system”, system_prompt), MessagesPlaceholder(“chat_history”, optional=True), (“human”, human_prompt),])

Powered by Granite

Now, lets finalize our prompt template by adding the tool names, descriptions and arguments using a partial prompt template. This allows the agent to access the information pertaining to each tool including the intended use cases. This also means we can add and remove tools without altering our entire prompt template.
prompt = prompt.partial(tools=render_text_description_and_args(list(tools)), tool_names=", ".join([t.name for t in tools]),)
Powered by Granite

Step 8. Set up the agent's memory and chain

An important feature of AI agents is their memory. Agents are able to store past conversations and past findings in their memory to improve the accuracy and relevance of their responses going forward. In our case, we will use LangChain's ConversationBufferMemory() as a means of memory storage.
memory = ConversationBufferMemory()
Powered by Granite

And now we can set up a chain with our agent's scratchpad, memory, prompt and the LLM. The AgentExecutor class is used to execute the agent. It takes the agent, its tools, error handling approach, verbose parameter and memory.
chain = (RunnablePassthrough.assign(agent_scratchpad=lambda x: format_log_to_str(x["intermediate_steps"]), chat_history=lambda x: memory.chat_memory.messages,) | prompt | llm | JSONAgentOutputParser()) agent_executor = AgentExecutor(agent=chain, tools=tools, handle_parsing_errors=True, verbose=True, memory=memory)
Powered by Granite

Step 9. Generate responses with the agentic RAG system

We are now able to ask the agent questions. Recall the agent's previous inability to provide us with information pertaining to the 2024 US Open. Now that the agent has its RAG tool available to use, let's try asking the same questions again.
agent_executor.invoke({"input": 'Where was the 2024 US Open Tennis Championship?'})
Powered by Granite

Output: (some description and page content fields were shortened to succinctly display results)

```
> Entering new AgentExecutor chain...

Thought: The human is asking about the location of the 2024 US Open Tennis Championship. I need to find out where it was held.
Action:
```
{
    "action": "get_IBM_US_Open_context",
    "action_input": "Where was the 2024 US Open Tennis Championship held?"
}
```
Observation[Document(metadata={'description': "IBM and the United States Tennis Association (USTA) announced several watsonx-powered fan

Action:
```
{
    "action": "Final Answer",
    "action_input": "The 2024 US Open Tennis Championship was held at the USTA Billie Jean King National Tennis Center in Flushing, Queen
}
```
Observation

> Finished chain.

{'input': 'Where was the 2024 US Open Tennis Championship?',
  'history': '',
  'output': 'The 2024 US Open Tennis Championship was held at the USTA Billie Jean King National Tennis Center in Flushing, Queens, New Y
```

Great! The agent used its available RAG tool to return the location of the 2024 US Open, per the user's query. We even get to see the exact document that the agent is retrieving its information from. Now, let's try a slightly more complex question query. This time, the query will be about IBM's involvement in the 2024 US Open.
agent_executor.invoke({"input": 'How did IBM use watsonx at the 2024 US Open Tennis Championship?'})
Powered by Granite

Output: (some description and page content fields were shortened to succinctly display results)

```
> Entering new AgentExecutor chain...
```
{
    "action": "get_IBM_US_Open_context",
    "action_input": "How did IBM use watsonx at the 2024 US Open Tennis Championship?"
}
```
Observation[Document(metadata={'description': 'To help the US Open stay on the cutting edge of customer experience, IBM Consulting built

Action:
```
{
    "action": "Final Answer",
    "action_input": "IBM used watsonx at the 2024 US Open Tennis Championship to create generative AI-powered features such as Match Repo
}
```
Observation

> Finished chain.

{'input': 'How did IBM use watsonx at the 2024 US Open Tennis Championship?',
 'history': 'Human: Where was the 2024 US Open Tennis Championship?\nAI: The 2024 US Open Tennis Championship was held at the USTA Billie
 'output': 'IBM used watsonx at the 2024 US Open Tennis Championship to create generative AI-powered features such as Match Reports, AI C
```

Again, the agent was able to successfully retrieve the relevant information pertaining to the user query. Additionally, the agent is successfully updating its knowledge base as it learns new information and experiences new interactions as seen by the history output.

Now, let's test if the agent can decipher when tool calling is not necessary to answer the user query. We can test this by asking the RAG agent a question that is not about the US Open.
agent_executor.invoke({"input": 'What is the capital of France?'})
Powered by Granite

Output:

```
> Entering new AgentExecutor chain...

{
    "action": "Final Answer",
    "action_input": "The capital of France is Paris."
}

Observation

> Finished chain.

{'input': 'What is the capital of France?',
 'history': 'Human: Where was the 2024 US Open Tennis Championship?\nAI: The 2024 US Open Tennis Championship was held at the USTA Billie
 'output': 'The capital of France is Paris.'}
```

As seen in the AgentExecutor chain, the agent recognized that it had the information in its knowledge base to answer this question without using its tools.

Summary

In this tutorial, you created a RAG agent using LangChain in python with watsonx. The LLM you worked with was the IBM Granite-3.2-8B-Instruct model. The sample output is important as it shows the significance of this generative AI advancement. The AI agent was successfully able to retrieve relevant information via the get_IBM_US_Open_context tool, update its memory with each interaction and output appropriate responses. It is also important to note the agent's ability to determine whether tool calling is appropriate for each specific task. When the agent had the information necessary to answer the input query, it did not use any tools for question answering.

For more AI agent content, we encourage you to check out our AI agent tutorial that returns today's Astronomy Picture of the Day using NASA's open source API and a date tool.