# Guide to Building and Running a Local RAG System (NBA Example)

## Introduction and Purpose

This guide explains how to set up and run a **Retrieval-Augmented Generation (RAG)** system using the provided project files (including Python scripts and a dataset). The goal is to enable a coworker to replicate the system step-by-step. We will cover:

* A brief overview of what RAG is and why it’s useful.
* Key components of the system (language model, embeddings, vector store, etc.).
* Installation and setup instructions on Windows (using PowerShell), including required tools and model downloads (with notes on model sizes and hardware considerations).
* Data ingestion process (using the NBA 2025 dataset as an example).
* Running the Q&A engine and example queries.

By the end of this document, you should be able to install the necessary software, ingest the data into a vector database, and ask questions that the local LLM can answer using the provided data. Code snippets and an NBA example are included for clarity.

## What is RAG (Retrieval-Augmented Generation)?

**Retrieval-Augmented Generation (RAG)** is a technique that combines a language model with an external knowledge source to improve the accuracy and relevance of generated answers. Instead of relying solely on what an AI model memorized during training, a RAG system *retrieves* relevant information from a document set (knowledge base) at query time and *augments* the model’s input with this context[[1]](https://en.wikipedia.org/wiki/Retrieval-augmented_generation#:~:text=Retrieval,responses%20based%20on%20authoritative%20sources)[[2]](https://en.wikipedia.org/wiki/Retrieval-augmented_generation#:~:text=on%20static%20training%20data%2C%20RAG,6). In practice, this means the model will consult a set of documents (for example, company files or a custom dataset) to find facts, and then use those facts to generate a grounded answer. This approach has several benefits:

* **Up-to-date and Domain-Specific Knowledge:** The model can access information not present in its original training data (e.g. recent or niche domain data)[[3]](https://en.wikipedia.org/wiki/Retrieval-augmented_generation#:~:text=training%20data%20.,responses%20based%20on%20authoritative%20sources). For example, an LLM can answer questions about the *2025 NBA season* by retrieving from a dataset of 2025 NBA articles, even if the LLM’s training cutoff was earlier.
* **Reduced Hallucinations:** By grounding responses in real documents, RAG helps the model stick to factual information and reduces the chance of it making things up[[2]](https://en.wikipedia.org/wiki/Retrieval-augmented_generation#:~:text=on%20static%20training%20data%2C%20RAG,6). Essentially, the model is *open-book*: it cites or uses the retrieved text rather than guessing.
* **No Need for Retraining the LLM:** We can update the document set (the knowledge base) as needed without retraining the large model. The LLM remains fixed, and any new information can simply be added to the index for retrieval[[4]](https://en.wikipedia.org/wiki/Retrieval-augmented_generation#:~:text=RAG%20also%20reduces%20the%20need,to%20ensure%20accuracy%20and%20relevance). This saves time and computational cost.
* **Transparency:** A RAG system can be configured to show the source documents or citations for its answers, so users can verify where the information came from (in our code, the RetrievalQA chain can return source documents for this purpose).

**How RAG Works (Key Steps):** At a high level, a RAG pipeline goes through the following stages:

1. **Indexing:** First, we process the knowledge documents by splitting them into chunks and converting those chunks into vector embeddings. These embeddings are stored in a *vector database* or index (allowing similarity search).
2. **Retrieval:** When a user asks a question, the system embeds the query into the same vector space and finds the most similar document chunks from the database (i.e. the pieces likely to contain relevant info).
3. **Augmentation:** The retrieved text chunks are then inserted into a prompt template as context.
4. **Generation:** The prompt (which now includes the context + question) is fed to the language model, which generates an answer. The model is instructed to use *only* the provided context to ensure factuality. If the context doesn’t contain the answer, it should reply with “I don’t know.” (Our prompt template explicitly handles this case).

In summary, *RAG enables an LLM to retrieve and incorporate new information from an external knowledge base before answering a query*, making it a powerful approach for custom Q&A systems[[1]](https://en.wikipedia.org/wiki/Retrieval-augmented_generation#:~:text=Retrieval,responses%20based%20on%20authoritative%20sources)[[2]](https://en.wikipedia.org/wiki/Retrieval-augmented_generation#:~:text=on%20static%20training%20data%2C%20RAG,6).

## System Components and Workflow

Our project implements a local RAG system using the following components:

* **Local Language Model (LLM):** We use the **Mistral 7B** model as the main large language model to generate answers. Mistral 7B is a 7.3 billion parameter open-source model known for its strong performance for its size[[5]](https://mistral.ai/news/announcing-mistral-7b#:~:text=Mistral%20AI%20team%20is%20proud,for%20its%20size%20to%20date). It’s provided under the Apache 2.0 license (free to use) and has an instruction-tuned variant suitable for Q&A or assistant tasks[[6]](https://mistral.ai/news/announcing-mistral-7b#:~:text=Mistral%207B%20is%20easy%20to,outperforms%20Llama%202%2013B%20chat). In this setup, the Mistral model is run locally via the **Ollama** backend. (*Ollama* is a tool that serves LLMs on your machine with an API; we’ll discuss installation shortly.) Mistral 7B, despite its relatively small parameter count, outperforms larger models like LLaMA-2 13B on many tasks[[7]](https://mistral.ai/news/announcing-mistral-7b#:~:text=Mistral%207B%20is%20a%207,parameter%20model%20that), making it a good choice for a limited hardware scenario.
* **Embedding Model:** We use **nomic-embed-text-v1**, an open-source text embedding model from Nomic AI, to convert texts (documents and queries) into high-dimensional vectors. This model has ~137 million parameters and supports long input texts (up to 8192 tokens)[[8]](https://huggingface.co/nomic-ai/nomic-embed-text-v1#:~:text=Safetensors)[[9]](https://huggingface.co/nomic-ai/nomic-embed-text-v1#:~:text=Model%20size). Despite being much smaller than the LLM, it produces embeddings that achieve state-of-the-art semantic search performance – it *even surpasses OpenAI’s text-embedding-ada-002 on many tasks*[[10]](https://huggingface.co/nomic-ai/nomic-embed-text-v1#:~:text=%60nomic,short%20and%20long%20context%20tasks). The embedding model runs locally (also via Ollama in our case), ensuring the whole pipeline is self-contained.
* **Vector Store (Database):** We use **Chroma** DB as the vector database to store document embeddings and enable similarity search. Chroma is an open-source vector store that can persist data to disk. In this project, after ingesting data, you’ll have a directory vectorstores/nba\_2025\_db/ containing the Chroma database (including a chroma.sqlite3 file and index files). LangChain’s Chroma integration is used to interface with this database. The vector store lets us quickly retrieve the top-$k$ relevant document chunks for any query based on cosine similarity of embeddings.
* **LangChain Framework:** The orchestration of the pipeline is handled by [LangChain](https://github.com/hwchase17/langchain) library, which provides convenient abstractions. We use LangChain’s RetrievalQA chain with our LLM and vector store. LangChain’s community extensions (the langchain\_community module) provide the Ollama integrations: OllamaEmbeddings for embedding generation and Ollama (or ChatOllama) for the LLM. The framework takes care of constructing the prompt with context (we supply a custom prompt template to enforce usage of context and proper behavior) and calling the LLM to get a result.
* **Local Data (NBA 2025 Dataset):** The knowledge source in our example is an Excel file nba\_2025\_rag\_dataset.xlsx containing information about the 2024-2025 NBA season (such as playoff results, awards, etc.). Each row in the spreadsheet appears to be an article or data point with fields like title, content, and source. This is the data we will ingest into the vector store. After ingestion, the system can answer NBA questions (from 2025) by pulling answers from this data. *For instance*, if asked *“Who was the MVP of the 2025 NBA Finals?”*, the system should retrieve the Finals article and answer with the name found (e.g. *“Shai Gilgeous-Alexander”*). If asked something unrelated or not covered, it should respond with “I don’t know.”

**Workflow Summary:** After installation, the process works as follows: (1) Run the ingestion script to load the Excel data, split it into chunks, and store embeddings in the Chroma DB. (2) Run the RAG Q&A script, which waits for user questions. (3) For each question, the appropriate vector store is selected (in our code, we infer the topic by keywords; e.g., any query containing "NBA" will use the nba\_2025\_db vector store). The question is cleaned and converted to an embedding using the same embedder. (4) Top relevant document chunks are retrieved. (5) These chunks are inserted into the prompt template along with the question, and the LLM (Mistral 7B) is invoked to generate an answer. (6) The answer is printed to the console. Optionally, one could also print the source documents for verification (the code has a section to do this, currently commented out for brevity).

The diagram below illustrates the RAG pipeline in this project:

* **User Query** → (Embed query) → **Retriever (Chroma DB)** finds relevant text chunks → those chunks are fed into **LLM (Mistral 7B)** with the query to generate → **Answer** (grounded in the retrieved data).

Next, we’ll cover how to set up everything step-by-step on a Windows machine.

## Installation and Setup (Windows/PowerShell)

Setting up the environment involves installing Python libraries, the Ollama engine, and downloading the models. The instructions below assume you are using **Windows** (PowerShell or Command Prompt) and have an internet connection to download dependencies. (If performing offline installation, you’d need to acquire the packages and models separately.)

**1. Install Python 3.** If you don’t have Python 3 installed, download it from the official site and install it (ensure you add Python to your PATH during installation). You can verify installation by opening PowerShell and running: python --version. The project is compatible with Python 3.8+.

**2. Set up a virtual environment (optional but recommended).** This helps keep project dependencies isolated. In PowerShell:

cd C:\path\to\project\folder # navigate to the project directory  
python -m venv .venv # create a virtual environment named .venv  
.\.venv\Scripts\Activate.ps1 # activate the virtual environment

After activation, your prompt will show (.venv) indicating the venv is active. (On CMD, the activation command would be .\.venv\Scripts\activate.bat.)

**3. Install required Python libraries.** With the venv activated (or in your global env if you skipped step 2), install the project’s dependencies using pip:

pip install pandas langchain chromadb ftfy

* pandas is used for reading the Excel dataset.
* langchain is the core framework (it will likely pull in langchain-ollama and other needed sub-packages automatically, but if not, we address that below).
* chromadb is the vector database library used by LangChain’s Chroma integration.
* ftfy is a small utility to fix text encoding issues (used when cleaning text).

*Note:* Newer versions of LangChain have factored some integrations into separate packages. If you see warnings or issues about Ollama or Chroma being deprecated in langchain, you may install their new packages: pip install langchain-ollama langchain-chroma. In our testing (LangChain v0.3.x), using the classes via langchain\_community still works but prints deprecation warnings. It’s okay to proceed despite the warnings, but in future, the code might need slight modifications (importing from langchain\_ollama etc.).

**4. Install Ollama (LLM runtime).** **Ollama** is the backend that runs the local LLM and embedding model. Download the **Ollama for Windows** installer from the official website (they have a Windows preview version) and run it[[11]](https://ollama.com/download/windows#:~:text=Download%20Ollama%20on%20Windows%20Download,get%20notified%20of%20new%20updates). The installer will set up the Ollama service and command-line tool. Once installed, ensure that the ollama command is accessible in PowerShell (you might need to open a new PowerShell or add it to PATH as prompted by the installer). Test by running:

ollama --version

which should display the version (e.g., ollama 0.1.x) if installed correctly.

Ollama provides a local model registry and API. We will use it to download the Mistral 7B model and the Nomic embed model next.

**5. Download the language model (Mistral 7B) via Ollama.** In PowerShell, run:

ollama pull mistral

This will download the **Mistral 7B** model (the instruction-tuned variant) to your local machine. The download is a few gigabytes (around 4 GB for the quantized version). You can monitor the progress as it pulls the model. Once done, you can run ollama list to verify the model is installed – it should list mistral:latest (with size ~4.1 GB).

**Model size and hardware note:** The base Mistral 7B model is ~13 GB in FP16 precision, but the version Ollama provides by default is likely 4-bit quantized (~4 GB file) for efficiency[[12]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=,Min). Mistral 7B is designed to be efficient and can run on mid-range GPUs or even CPU (with performance trade-offs). In fact, a 4-bit quantized 7B model can often fit in ~4–6 GB of GPU VRAM or in about ~8+ GB of system RAM for CPU inference[[13]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=,Min). If you have a GPU with at least 6–8 GB VRAM, you should be able to run it with decent speed. With no GPU, the model will run on CPU – which is **possible but slow** (expect on the order of minutes per response on a typical PC)[[14]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=3,Is%20It%20Possible)[[15]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=Mistral%207B%20is%20one%20of,quickly%20without%20breaking%20the%20bank).

*If you are very limited in hardware (e.g., no strong GPU and low RAM):* you have a few options: (a) stick with the 4-bit quantized Mistral and accept slower responses, (b) try an even more quantized variant if available (Ollama might allow mistral:7b-q8 etc., though 4-bit is already quite compact), or (c) use a smaller model entirely (e.g., a 3–5B parameter model) at the cost of answer quality[[16]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=,focused). Option (c) might not be necessary since Mistral 7B is one of the best trade-offs in size vs performance[[15]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=Mistral%207B%20is%20one%20of,quickly%20without%20breaking%20the%20bank) – it’s highly recommended if you can manage it. (For reference, Mistral 7B often outperforms older 13B models, so dropping to a 3B model would significantly reduce answer accuracy[[5]](https://mistral.ai/news/announcing-mistral-7b#:~:text=Mistral%20AI%20team%20is%20proud,for%20its%20size%20to%20date)[[7]](https://mistral.ai/news/announcing-mistral-7b#:~:text=Mistral%207B%20is%20a%207,parameter%20model%20that).) In summary, use Mistral 7B if possible, and utilize the quantized version to keep memory usage low.

**6. Download the embedding model (Nomic Embed Text) via Ollama.** Next, run:

ollama pull nomic-embed-text

This will download the **nomic-embed-text-v1** model, used for generating text embeddings. This model is much smaller than Mistral – roughly a few hundred megabytes (as a 137M parameter model, ~0.5 GB in float32)[[9]](https://huggingface.co/nomic-ai/nomic-embed-text-v1#:~:text=Model%20size)[[17]](https://huggingface.co/nomic-ai/nomic-embed-text-v1#:~:text=137M%20params). Once downloaded, verify with ollama list that nomic-embed-text:latest appears. Ollama will handle serving this model for embedding requests.

*About this embedding model:* Nomic’s embed model is a state-of-the-art open embedding model with a long context (8192 tokens) and high accuracy on retrieval tasks[[10]](https://huggingface.co/nomic-ai/nomic-embed-text-v1#:~:text=%60nomic,short%20and%20long%20context%20tasks). It’s a suitable replacement for proprietary embeddings like OpenAI’s Ada-002. This means our RAG system will have vector search quality on par with or better than Ada, entirely offline.

**7. Verify all components are ready:** You should now have the Python environment set up with LangChain + dependencies, and the Ollama service running with the two models (Mistral and Nomic) available. Double-check: in PowerShell, you can run ollama list and expect output like:

NAME | SIZE | STATUS   
mistral:latest | 4.1 GiB | ready  
nomic-embed-text:latest | 500 MiB | ready

*(Sizes are examples; exact values may vary.)* If the status is not “ready” or you encounter issues, ensure that the Ollama service is running. On Windows, Ollama might run as a background service accessible via the CLI. You can also test an embedding via Python just to ensure connectivity, but that’s optional – the main test will be during ingestion.

With installation complete, we can ingest the dataset and then run the Q&A engine.

## Data Ingestion Process (Preparing the Vector Store)

Ingestion refers to converting our raw data (the Excel sheet of NBA info) into a form that the system can use for retrieval. The provided script ingest\_nba\_2025.py automates this. Let’s go through what it does and how to run it:

**Dataset:** nba\_2025\_rag\_dataset.xlsx – ensure this file is present in the same directory as the scripts. It contains NBA 2025 season info. The script uses pandas to read it. No special preprocessing is needed beyond what the script does, but if the file were in a different location, you’d need to adjust the EXCEL\_PATH in the script or move the file accordingly.

**What the ingest script does (steps):**

1. **Load the Excel file.** The script uses pandas to read all rows of the spreadsheet into a DataFrame. Each row corresponds to an article or piece of content. (For example, one row might be the 2025 NBA Finals recap, another might be about playoff statistics, etc.)
2. **Combine columns into text.** The Excel has columns like “title”, “content”, “source”, etc. The script iterates through each row and concatenates all non-empty cells into one text block. It does this by joining f"{column\_name}: {value}" for each column. The result for each row is a single string containing that row’s title, content, and any source URL or extra fields, separated by newlines. This becomes the raw text of a document. For instance, a row might yield:

title: 2025 NBA Finals: Thunder defeat Pacers in seven games  
content: The 2025 NBA Finals matched the top-seeded Oklahoma City Thunder with ...  
source: https://www.nba.com/news/2025-nba-playoffs-schedule#... (etc.)

Each such string is then wrapped in a LangChain Document object (which simply holds the text and maybe metadata, though we aren’t adding special metadata here).

1. **Split into chunks.** We don’t want to store extremely large text blobs in the vector DB, as searching works better on smaller, coherent pieces of text. The script uses LangChain’s RecursiveCharacterTextSplitter to break each document into chunks of up to 400 characters, with a 80-character overlap between chunks. This ensures that even if an answer spans a boundary, it might still be caught in an overlapping chunk. After this step, a single long article might turn into 2–3 chunks, while shorter entries might remain as one chunk. In total, our NBA dataset (35+ rows of data) resulted in **116 chunks** being generated and stored (the script prints the count at the end). Each chunk is a Document ready to be embedded.
2. **Embed the chunks.** The script initializes an embedding model instance:

embedding = OllamaEmbeddings(model="nomic-embed-text")

This uses the Ollama backend to generate embeddings for each text chunk. The model “nomic-embed-text” produces a numeric vector (likely 768 dimensions) for each chunk that captures its semantic meaning.

1. **Store in Chroma vector database.** The script then creates a Chroma vector store from these documents:

vectordb = Chroma.from\_documents(documents=split\_docs,   
 embedding=embedding,   
 persist\_directory=VSTORE\_DIR)  
vectordb.persist()

This call computes and stores all embeddings in a local database under VSTORE\_DIR (which is set to "vectorstores/nba\_2025\_db"). The persist() ensures the data is saved to disk for reuse. After running the script, the vectorstores/nba\_2025\_db/ folder will contain the Chroma index (you’ll see files like chroma.sqlite3, etc.). We will load this database later when we run the Q&A engine.

1. **Completion message:** The script prints a success message, e.g. ✅ Ingested 116 documents to vectorstores/nba\_2025\_db. This tells you it worked.

**Running the ingest script:** Open PowerShell in the project directory and run:

python ingest\_nba\_2025.py

This will execute the above steps. The first time, it may take a few minutes because it has to generate all embeddings by calling the Ollama embed model for each chunk. (The nomic model is quite fast, but if there are ~100+ chunks, it will still take some seconds – you’ll see the script working.) If everything is set up correctly, you should see the ✅ message at the end.

If you encounter any errors here, common issues could be: the Excel file not found (ensure path is correct), Ollama not running or the model names not found (make sure ollama pull was done and that you have the latest Ollama which supports these models), or memory issues (unlikely with embedding). The LangChain/Ollama integration will handle communicating with the Ollama service – you might see it starting the nomic-embed-text model on first use.

**Note:** The script as provided doesn’t encrypt or filter content; it simply takes raw text. The NBA dataset contained some Unicode characters (e.g., en-dashes, special quotes) – these should be handled fine, but we use ftfy.fix\_text in the query code to clean any odd characters in outputs. No action needed, just be aware if you use your own data, you might want to clean it similarly.

By the end of this ingestion step, we have our *knowledge base* ready: the NBA data is embedded and indexed in nba\_2025\_db. We can now query it using the RAG chain.

## Running the Q&A Engine (Retrieval + Generation)

With the data ingested, you can now launch the interactive question-answering engine. The script for this is rag\_engine.py. This script sets up the RetrievalQA chain and then enters a loop waiting for user input. Let’s break down what it does and how to use it:

**How rag\_engine.py works:**

* **Selecting the Vector Store:** The script has a function infer\_topic\_from\_query(query) that checks the user’s question for certain keywords to decide which vectorstore (database) to use. In our case, it looks for "nba" or related terms to pick the nba\_2025\_db. (There are placeholders for "nfl" or "pokemon" topics as well – presumably the project could be extended with other datasets. If a query doesn’t match any known topic, it defaults to "default" which would be another vectorstore.) This means if you ask anything about the NBA or finals, it will direct the system to use the NBA 2025 data we ingested. This is a simple routing mechanism. If you only have the NBA dataset, any question you ask should ideally contain “NBA” or a team/player name so that it selects the right DB. (You can modify this logic or hardcode topic = "nba\_2025\_db" if you know you’ll only use that dataset.)
* **Prompt Template:** A custom prompt is defined to guide the LLM’s behavior. The template (in LangChain’s PromptTemplate.from\_template syntax) is:

You are a factual Q&A assistant. Use ONLY the provided context to answer the question below.  
  
If the answer is clearly stated, return it exactly.   
If the answer is only implied, make a reasoned guess based on evidence.   
If the answer is not present, say: "I don't know."  
  
Context:  
---------  
{context}  
---------  
  
Question: {question}  
  
Answer:

This prompt ensures the model knows it must stick to the given context and not hallucinate. It also sets up the format by providing the context snippet(s) and the user’s question, prompting for an Answer. This will be used with a “stuff” chain type, meaning all retrieved documents will be simply concatenated into the {context} slot.

* **Initializing the LLM and Retriever:** The code then does the following when you call the main query\_rag(user\_query) function:

topic = infer\_topic\_from\_query(user\_query)  
db\_path = os.path.join("vectorstores", topic)  
  
if not os.path.exists(db\_path):  
 return f"Error: No vectorstore found for topic '{topic}'..."  
  
llm = Ollama(model="mistral")  
embedding = OllamaEmbeddings(model="nomic-embed-text")  
vectordb = Chroma(persist\_directory=db\_path, embedding\_function=embedding)  
retriever = vectordb.as\_retriever(search\_kwargs={"k": 6})  
qa\_chain = RetrievalQA.from\_chain\_type(  
 llm=llm,  
 retriever=retriever,  
 chain\_type="stuff",  
 chain\_type\_kwargs={"prompt": base\_prompt},  
 return\_source\_documents=True  
)  
result = qa\_chain({"query": user\_query})  
return result

A few things to note here:  
- It loads the Chroma DB from disk (persist\_directory) and uses the same embedding function for consistency. This allows it to perform similarity search.  
- search\_kwargs={"k": 6} means it will retrieve up to 6 chunks most relevant to the query. We chose 6 somewhat arbitrarily; you can adjust this. In practice, the top 2–3 might already have the answer, but we include more to be safe (too many can risk diluting the prompt or hitting token limits; 6 chunks of ~400 chars each is about a few thousand tokens at most, which is fine for Mistral 7B’s context window).  
- We pass our prompt template to the chain, and also request return\_source\_documents=True. The chain’s output will then be a dict with result (the answer string) and source\_documents (the actual Document objects retrieved). In our interactive loop, we print only the answer by default, but we could also print sources for verification or debugging.

* **Interactive Loop:** When you run rag\_engine.py as a script (i.e., if \_\_name\_\_ == "\_\_main\_\_": part), it enters a while True loop: it prompts you for a question, you type something, and it processes it. If you type "exit", the loop breaks and the program ends. For each query: it calls query\_rag(query). The result from query\_rag could be an error string (if no vectorstore found), or the normal output dict. If it’s an error (string), the script prints it with a ❌. If it’s a normal result, it prints the answer under a heading "📘 Answer:". The code has lines (commented out) to also print the source documents used, labeled [Document 1], [Document 2], etc., showing the first 300 characters of each. This is very useful for debugging or ensuring the model had the right info, so you may uncomment those lines if you want to see what context was retrieved each time. By default, our guide assumes we focus on the answer.

**Running the Q&A script:** In PowerShell, run:

python rag\_engine.py

You should see it print the prompt Ask a question (or type 'exit'):. Now you can type in a query. Let’s try an example relevant to our dataset:

* **Example query:** Who won the 2025 NBA Finals? – After you press Enter, the system will: detect “NBA” in your question and choose the nba\_2025\_db; embed your question; retrieve chunks (likely the chunk that mentions the Finals result); and feed them to Mistral. Ideally, the context includes a line like “OKC won the series 4-3 for its first championship...” from the Finals article. The model should then answer: *“The Oklahoma City Thunder won the 2025 NBA Finals.”* (If the context wasn’t sufficient or the model was cautious, it might say it cannot find the info – but since the data does have that, it should answer correctly. If you find it says “I don’t know” even though the info is there, you might need to tweak the question wording or verify the data ingestion. In our tests, the model was sometimes a bit literal with the “use ONLY context” rule and if the context phrasing didn’t explicitly say “Thunder won”, it might have hesitated. But generally it should get it from “Thunder ... for its first championship”.)
* **Example query 2:** Name the 16 teams that were in the 2025 NBA playoff bracket. – This is a more complex question that expects the list of all playoff teams. The data might have that in a particular article or might require combining info from multiple rows (like standings or bracket info). The retriever will pull possibly a chunk that lists playoff matchups. The model will try to enumerate the teams. This tests the system’s ability to handle a broader question. If the info is fragmented, the model might list what it found or say it’s not fully in context. Remember, the model is bound to the provided context – it won’t generate teams that are not mentioned in the retrieved text. So if it doesn’t retrieve all 16 team names, it might only list some or say it’s unsure. This is expected behavior in RAG: the answer is only as complete as the documents you provide. In any case, it shows how the system works with the given data.
* **Example query 3:** Who was the MVP of the 2025 Finals and what was his points average? – This should retrieve the Finals recap which mentions *“Gilgeous-Alexander captured the Finals MVP after averaging 30.3 points…”*. The model can then answer: “Shai Gilgeous-Alexander was the Finals MVP, and he averaged 30.3 points.” This demonstrates pulling a specific stat from the context.

Feel free to ask any question related to the 2025 NBA season, especially focusing on playoffs and finals, since the dataset seems centered on that. If you ask something completely unrelated (e.g. “Who won the 2022 World Cup?”), the system might either route to a “default” (which we didn’t populate) and give an error, or simply not find anything and respond with "I don't know." The design here expects you to ask about NBA or other integrated topics.

**Stopping the program:** To exit the Q&A loop, type exit and press Enter. That will break the loop and end the script.

## Model Size and Performance Considerations

Because we’re running everything locally, it’s important to understand the resource usage:

* **Mistral 7B (LLM):** As discussed, the quantized model is about 4 GB on disk. In memory, Ollama will load this model – likely also around 4–5 GB RAM usage (plus some overhead). If you have a GPU, it may use the GPU memory for acceleration. Mistral 7B is one of the most efficient models for its performance; it can often run on consumer GPUs (NVIDIA 30-series or better) and even on CPU with enough RAM[[18]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=8). If your coworker’s machine is very limited (say 8GB RAM total and integrated graphics), running Mistral might push memory limits. In such case, closing other applications and ensuring a large swap file could help, or opting for a smaller model as noted. However, the quality drop is significant if you go much smaller, so ideally they should run Mistral if at all possible. Another option is to run the model in 4-bit mode using a library like llama-cpp-python, but since we’re using Ollama, it already handles an optimized runtime.
* **Nomic Embed model:** At ~137M params (0.5GB), this is lightweight. It will also use some CPU/GPU but typically generating embeddings is fast (especially since it’s likely a transformer-based encoder). It won’t be the bottleneck in our pipeline.
* **Chroma DB:** This runs in-process (via the Python code). For ~116 embeddings of 768 dims, this is negligible memory. Chroma might spin up a SQLite connection, etc., but it’s minimal. So no concerns here.
* **LangChain overhead:** Minimal, just Python logic.
* **Concurrency:** Our current setup is single-user, single-query at a time (the script loop). If one wanted to build a server or handle multiple queries concurrently, they’d have to manage the calls to Ollama (which does support an API) and perhaps run the LLM in parallel, but that’s beyond this use-case. For a single user Q&A, the interactive loop is fine.

**Performance:** The response time per query will depend on hardware: On a decent GPU, Mistral 7B can generate answers in a couple of seconds. On CPU, it might take 30 seconds or more for a response (especially if the answer is long). The embedding retrieval part is very fast (<1s). The bulk is the LLM generation. So if you notice a delay after you hit Enter on a question, that’s normal – it’s the model thinking. If it’s too slow, consider the suggestions above (ensure it’s using GPU if available, or in worst case, try a smaller model or only ask for short answers).

## Summary and Next Steps

We have set up a local RAG system that can answer questions about the 2025 NBA season by referencing the provided data. To recap the **workflow**: we installed the environment, ingested the data into a vector store, and then used a RetrievalQA chain (with **Mistral 7B** as the brain and the NBA articles as the knowledge) to answer questions. This approach demonstrates how you can **give a large language model an external “knowledge base” and update that knowledge at any time** without retraining the model – a very powerful pattern.

With this foundation, you or your coworkers can extend the system by: adding more data (perhaps other sports or topics, ingested into their own vectorstores), improving the prompt or chain type (e.g., using a map-reduce chain for long answers, though “stuff” worked fine here), or even swapping in a different model if needed. For instance, if down the line a more powerful model that can still run on your hardware becomes available, you can ollama pull it and change the llm = Ollama(model="...") line to use it. Just be mindful of the hardware constraints and model sizes discussed.

Finally, to ensure everything is reproducible: all steps above can be done on another Windows machine following the guide. If any issues arise during replication, double-check each installation step (especially Ollama and model downloads) and confirm file paths. With the detailed breakdown provided, a coworker should be able to understand *how* the system works and not just run it, but also modify it for their own purposes.

Good luck, and enjoy your fully local Q&A system powered by RAG! If you encounter any problems or have questions about the setup, refer back to this guide or the cited resources for troubleshooting.

[[1]](https://en.wikipedia.org/wiki/Retrieval-augmented_generation" \l ":~:text=Retrieval,responses%20based%20on%20authoritative%20sources) [[2]](https://en.wikipedia.org/wiki/Retrieval-augmented_generation#:~:text=on%20static%20training%20data%2C%20RAG,6) [[3]](https://en.wikipedia.org/wiki/Retrieval-augmented_generation#:~:text=training%20data%20.,responses%20based%20on%20authoritative%20sources) [[4]](https://en.wikipedia.org/wiki/Retrieval-augmented_generation#:~:text=RAG%20also%20reduces%20the%20need,to%20ensure%20accuracy%20and%20relevance) Retrieval-augmented generation - Wikipedia

<https://en.wikipedia.org/wiki/Retrieval-augmented_generation>

[[5]](https://mistral.ai/news/announcing-mistral-7b#:~:text=Mistral%20AI%20team%20is%20proud,for%20its%20size%20to%20date) [[6]](https://mistral.ai/news/announcing-mistral-7b#:~:text=Mistral%207B%20is%20easy%20to,outperforms%20Llama%202%2013B%20chat) [[7]](https://mistral.ai/news/announcing-mistral-7b#:~:text=Mistral%207B%20is%20a%207,parameter%20model%20that) Mistral 7B | Mistral AI

<https://mistral.ai/news/announcing-mistral-7b>

[[8]](https://huggingface.co/nomic-ai/nomic-embed-text-v1#:~:text=Safetensors) [[9]](https://huggingface.co/nomic-ai/nomic-embed-text-v1#:~:text=Model%20size) [[10]](https://huggingface.co/nomic-ai/nomic-embed-text-v1#:~:text=%60nomic,short%20and%20long%20context%20tasks) [[17]](https://huggingface.co/nomic-ai/nomic-embed-text-v1#:~:text=137M%20params) nomic-ai/nomic-embed-text-v1 · Hugging Face

<https://huggingface.co/nomic-ai/nomic-embed-text-v1>

[[11]](https://ollama.com/download/windows#:~:text=Download%20Ollama%20on%20Windows%20Download,get%20notified%20of%20new%20updates) Download Ollama on Windows

<https://ollama.com/download/windows>

[[12]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=,Min) [[13]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=,Min) [[14]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=3,Is%20It%20Possible) [[15]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=Mistral%207B%20is%20one%20of,quickly%20without%20breaking%20the%20bank) [[16]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=,focused) [[18]](https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs#:~:text=8) Mistral 7B System Requirements: What You Need to Run It Locally

<https://www.oneclickitsolution.com/centerofexcellence/aiml/run-mistral-7b-locally-hardware-software-specs>