Step 1:

Avoid this step if you are using Google Collab

1. Set up virtual environment

   pip install virtualenv
   python -m venv env

2. Activate Command

   Activation Command: .\env\Scripts\activate
   Command to deactivate virtual environment: deactivate

Step 2:

1. If using google collab — Upload pdf file to contents
2. If using IDE — Create ‘Data’ folder and add pdf file to it

Step 3:

Avoid this step if you are using Google Collab

1. Create .gitignore file — to avoid pushing content of codebase on version control / cloud
2. Create .env file — to store API keys and private credentials
3. Add virtual environment and env file to .gitignore — because
   a. Virtual environment is needed for coding locally
   b. .env has private credentials and we aren’t in favour of pushing such data on cloud

Step 4:

1. pip install commands for all required packages

   pip install langchain-community
   pip install langchain-text-splitters
   pip install langchain-huggingface
   pip install langchain-chroma
   pip install langchain-groq
   pip install langchain-core
   pip install langchain-classic
   pip install sentence-transformers
   pip install pypdf
   pip install python-dotenv

2. single combined command

   pip install langchain-community langchain-text-splitters langchain-huggingface langchain-chroma langchain-groq langchain-core langchain-classic sentence-transformers pypdf python-dotenv

Step 5: Load Data

```
from langchain_community.document_loaders import PyPDFLoader
loader = PyPDFLoader("data/Rudalph_Resume.pdf")
data = loader.load()
```

a. This code imports a tool that can read PDF files.
b. It creates a PDF loader for the file Rudalph_Resume.pdf stored in the data folder.
c. It loads the PDF content and saves it in the variable data.

Step 6: Split text into chunks

```
from langchain_text_splitters import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500)
docs = text_splitter.split_documents(data)
```

a. This imports a tool that breaks big text into smaller parts.
b. It creates a splitter that cuts text into chunks of 500 characters.
c. It splits the loaded PDF data into smaller pieces and stores them in docs.
d. We do this so the AI can understand and search the text better, because smaller chunks improve accuracy and reduce memory load.

Step 7: Creating instance of embedding model

```
from langchain_huggingface import HuggingFaceEmbeddings
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
```

a. This imports a tool that converts text into numerical vectors (embeddings).
b. It loads a HuggingFace embedding model called "all-MiniLM-L6-v2".
c. We do this so the AI can compare text pieces and find the most relevant information during search.

Step 8: Create Vector Store

```
from langchain_chroma import Chroma
vectorstore = Chroma.from_documents(
    documents=docs,
    embedding=embeddings
)
```

a. This imports Chroma, a database that stores text as vectors.
b. It creates a vectorstore by saving all document chunks (docs) in Chroma.
c. It uses the embeddings model to convert each chunk into vectors before storing.
d. We do this so the AI can quickly search and retrieve the most relevant text based on meaning, not exact words.

Step 9: Creating retriever's instance

```
retriever = vectorstore.as_retriever(search_type="similarity")
```

a. This creates a **retriever**, a tool that searches the vector database.
b. as_retriever() converts the vectorstore into a search engine.
c. search_type="similarity" means it finds chunks that have the closest meaning to the question.
d. We do this so the AI can pull the most relevant text from your documents.

Step 10: Creating LLM's instance

```
from dotenv import load_dotenv
import os
from langchain_groq import ChatGroq
load_dotenv()
llm_api_key = os.getenv("GROQ_API_KEY")
llm=ChatGroq(groq_api_key=llm_api_key,model_name="llama-3.1-8b-instant"
, temperature=0.5)
```

a. This imports tools to load environment variables and use the Groq LLM.
b. load_dotenv() reads your .env file so your API key becomes available.
c. os.getenv("GROQ_API_KEY") fetches the API key safely from the environment.
d. ChatGroq(...) creates an LLM (AI model) using LLaMA 3.1 with your key.
e. We do this so the AI can answer questions using the Groq-powered LLM.

Step 11: User's query

```
user_query = input("Enter your question: ")
```

a. This line asks the user to type a question in the terminal.

b. Whatever the user types is stored in the variable user_query.
c. We do this so the AI knows what question it needs to answer.

Step 12: Prompts and Prompt Templates

```
from langchain_core.prompts import ChatPromptTemplate
system_prompt = (
    "You are an assistant for question-answering tasks. "
    "Use the following pieces of retrieved context to answer "
    "the question. If you don't know the answer, say that you "
    "don't know. Use three sentences maximum and keep the "
    "answer concise.\n\n{context}"
)
prompt = ChatPromptTemplate.from_messages([
    ("system", system_prompt),
    ("human", "{input}"),
])
```

a. This imports a tool to create structured prompts for the AI.
b. system_prompt tells the AI how to behave, use context, and keep answers short.
c. {context} is where retrieved document chunks go.
d. ChatPromptTemplate.from_messages combines system instructions and user input into a chat-ready prompt.
e. We do this so the AI answers accurately and concisely using the documents.

Step 13: Developing RAG Chains

```
from langchain_classic.chains import create_retrieval_chain
from langchain_classic.chains.combine_documents import
create_stuff_documents_chain

if user_query:
    question_answer_chain = create_stuff_documents_chain(llm, prompt)
    rag_chain = create_retrieval_chain(retriever,
question_answer_chain)
    response = rag_chain.invoke({"input": user_query})
    print(response["answer"])
```

a. This imports tools to create RAG chains for question-answering.
b. create_stuff_documents_chain(llm, prompt) sets up a chain that uses the LLM and prompt to answer questions.
c. create_retrieval_chain(retriever, question_answer_chain) combines the retriever and the LLM chain into a RAG chain.
d. rag_chain.invoke({"input": user_query}) runs the user's question through the RAG system.
e. print(response["answer"]) shows the AI's final answer to the user.

Step 14: Final Output