Course 7 - Cheat Sheet: Project: Generative AI Applications with RAG and LangChain

Package/ Method	Description	Code Example
Load method	Loads data from a server and puts the returned data into the selected element.	`data = loader.load()`
Document object	Contains information about some data in LangChain. It has two attributes: • page_content: str: This attribute holds the content of the document. • metadata: dict: This attribute contains arbitrary metadata associated with the document. It can be used to track various details such as the document id, file name, and so on.	`from langchain_core.documents import Document` `Document(page_content="""Python is an interpreted high-level general-purpose programming language.` `                        Python's design philosophy emphasizes code readability with its notable use of significant indentation.""",` `         metadata={` `             'my_document_id' : 234234,` `             'my_document_source' : "About Python",` `             'my_document_create_time' : 1680013019` `         })`
pprint function	A function in Python used to “pretty-print” data structures, making them more readable and easier to understand.	`pprint(data[0].page_content[:1000])`
PyPDFLoader	Simplifies the process of loading PDF documents into a format that can be easily manipulated and analyzed within your applications.	`pdf_url = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/Q81D33CdRLK6LswuQrANQQ/instructlab.pdf"` `loader = PyPDFLoader(pdf_url)`

<table>
<tr><td></td><td></td><td><pre><code>pages = loader.load_and_split()</code></pre></td></tr>
<tr><td>PyMuPDFLoader</td><td>The fastest of the PDF parsing options. It provides detailed metadata about the PDF and its pages and returns one document per page.</td><td><pre><code>loader = PyMuPDFLoader(pdf_url)
loader

data = loader.load()

print(data[0])</code></pre></td></tr>
<tr><td>UnstructuredMarkdownLoader</td><td>A powerful tool within the Langchain framework that facilitates the loading of Markdown documents into a structured format suitable for downstream processing.</td><td><pre><code>!wget 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/eMSP5vJjj9yOfAacLZRWsg/markdown-sample.md'

markdown_path = "markdown-sample.md"
loader = UnstructuredMarkdownLoader(markdown_path)
loader

data = loader.load()

data</code></pre></td></tr>
<tr><td>JSONLoader</td><td>A module that builds a straightforward Python object from loaded JSON or similar dict-based data loading. It also checks if the input-loaded JSON has all the necessary attributes for the pipeline and that it has the right types.</td><td><pre><code>!wget 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/hAmzVJeOUAMHzmhUHNdAUg/facebook-chat.json'</code></pre></td></tr>
<tr><td>CSVLoader</td><td>CSV files are a common format for storing tabular data. The CSVLoader provides a convenient way to read and process this data.</td><td><pre><code>!wget 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IygVG_j0M87BM4Z0zFsBMA/mlb-teams-2012.csv'

loader = CSVLoader(file_path='mlb-teams-2012.csv')
data = loader.load()</code></pre></td></tr>
</table>

<table>
<tr><td></td><td></td><td><pre><code>data</code></pre></td></tr>
<tr><td>UnstructuredCSVLoader</td><td>The UnstructuredCSVLoader considers the entire CSV file as a single unstructured table element. This approach is beneficial when you want to analyze the data as a complete table rather than as separate entries.</td><td><pre><code>loader = UnstructuredCSVLoader(
    file_path="mlb-teams-2012.csv", mode="elements"
)
data = loader.load()

data[0].page_content

print(data[0].metadata["text_as_html"])</code></pre></td></tr>
<tr><td>BeautifulSoup</td><td>A Python library used for web scraping purposes to pull the data out of HTML and XML files. It creates a parse tree for parsed pages that can be used to extract data easily.</td><td><pre><code>import requests
from bs4 import BeautifulSoup

url = 'https://www.ibm.com/topics/langchain'
response = requests.get(url)

soup = BeautifulSoup(response.content, 'html.parser')
print(soup.prettify())</code></pre></td></tr>
<tr><td>WebBaseLoader</td><td>LangChain's tool designed to extract all text from HTML webpages and convert it into a document format suitable for further processing.</td><td><pre><code>For single page:
loader =
WebBaseLoader("https://www.ibm.com/topics/langchain")

data = loader.load()

data

For multiple pages:
loader =
WebBaseLoader(["https://www.ibm.com/topics/langchain",
"https://www.redhat.com/en/topics/ai/what-is-
instructlab"])</code></pre></td></tr>
</table>

		`data = loader.load()` `data`
Docx2txtLoader	Utilized to convert Word documents into a document format suitable for further processing.	`!wget https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/94hiHUNLZdb0bLMkrCh79g/file-sample.docx` `loader = Docx2txtLoader("file-sample.docx")` `data = loader.load()` `data`
Load .txt file	Supports the loading of .txt files when you need to load content from various text sources and formats without writing a separate loader for each one.	`loader = UnstructuredFileLoader("companypolicies.txt")` `data = loader.load()` `data`
Load .md file	Supports the loading of .md files when you need to load content from various text sources and formats without writing a separate loader for each one.	`loader = UnstructuredFileLoader("markdown-sample.md")` `data = loader.load()` `data`
Load multiple files with different formats	Supports the loading of multiple file types when you need to load content from various text sources and formats without writing a separate loader for each one.	`files = ["markdown-sample.md", "companypolicies.txt"]` `loader = UnstructuredFileLoader(files)` `data = loader.load()` `data`
Model ID	In LangChain, the model ID is used to specify which language model you want to use. This ID can vary depending on the model provider	`def llm_model(model_id):` `    parameters = {` `        GenParams.MAX_NEW_TOKENS: 256,  # this controls the maximum number of tokens in the generated output`

	and the specific model you are accessing.	<pre> GenParams.TEMPERATURE: 0.5, # this randomness or creativity of the model's responses } credentials = { "url": "https://us-south.ml.cloud.ibm.com" } project_id = "skills-network" model = ModelInference(model_id=model_id, params=parameters, credentials=credentials, project_id=project_id) llm = WatsonxLLM(watsonx_model = model) return llm</pre>
Load source document	Loading a source document into a Large Language Model (LLM) involves providing the model with specific data or text that it can use to generate responses or perform tasks.	`!wget "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/d_ahNwb1L2duIxBR6RD63Q/state-of-the-union.txt"`
LangChain prompt template	A prompt template is set up using LangChain to make it reusable.	<pre>template = """According to the document content here {content}, answer this question {question}. Do not try to make up the answer. YOUR RESPONSE: """</pre>

		`prompt_template = PromptTemplate(template=template, input_variables=['content', 'question'])` `prompt_template`
Use mixtral model	A sparse mixture-of-experts (SMoE) network developed by Mistral AI. It is a decoder-only transformer model with a unique architecture that includes 8 experts per feedforward block, totaling 45 billion parameters.	`mixtral_llm = llm_model('mistralai/mixtral-8x7b-instruct-v01')` `query_chain = LLMChain(llm=mixtral_llm, prompt=prompt_template)` `query = "It is in which year of our nation?"` `response = query_chain.invoke(input={'content': content, 'question': query})` `print(response['text'])`
Use Llama 3 model	The Llama model (Large Language Model Meta AI) is a family of autoregressive large language models developed by Meta AI.	`query_chain = LLMChain(llm=llama_llm, prompt=prompt_template)` `query_chain`
Use one piece of information	Using this code snippet, retrieve one piece of information related to the query and put it in the content variable.	`content = """` `    The only nation that can be defined by a single word: possibilities.` `    So on this night, in our 245th year as a nation, I have come to report on the State of the Union.` `    And my report is this: the State of the Union is strong—because you, the American people, are strong.` `"""`
Split by Character	This is the simplest method of splitting text, which splits the text	`from langchain.text_splitter import CharacterTextSplitter`

<table>
<tr>
<td></td>
<td>based on characters (by default "\n\n") and measures chunk length by the number of characters.</td>
<td><pre><code>text_splitter = CharacterTextSplitter(
    separator="",
    chunk_size=200,
    chunk_overlap=20,
    length_function=len,
)</code></pre></td>
</tr>
<tr>
<td>Recursively Split by Character</td>
<td>A text splitter recommended for generic text. It is parameterized by a list of characters, and it tries to split them in order until the chunks are small enough.</td>
<td><pre><code>from langchain.text_splitter import
RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=100,
    chunk_overlap=20,
    length_function=len,
)</code></pre></td>
</tr>
<tr>
<td>Split Code</td>
<td>This method allows you to split your code, supporting multiple programming languages. It is based on the Recursively Split by Character strategy.</td>
<td><pre><code>PYTHON_CODE = """
    def hello_world():
        print("Hello, World!")

    # Call the function
    hello_world()
"""
python_splitter =
RecursiveCharacterTextSplitter.from_language(
    language=Language.PYTHON, chunk_size=50,
chunk_overlap=0
)
python_docs =
python_splitter.create_documents([PYTHON_CODE])
python_docs</code></pre></td>
</tr>
</table>

Markdown Header Text Splitter	A Markdown file is organized by headers. Creating chunks within specific header groups is an intuitive approach. This splitter will divide a Markdown file based on a specified set of headers.	`markdown_splitter = MarkdownHeaderTextSplitter(headers_to_split_on=headers_to_split_on)` `md_header_splits = markdown_splitter.split_text(md)` `md_header_splits`
Split by HTML	This splitting method is a "structure-aware" chunker that splits text at the element level and adds metadata for each header "relevant" to any given chunk.	`html_splitter = HTMLHeaderTextSplitter(headers_to_split_on=headers_to_split_on)` `html_header_splits = html_splitter.split_text(html_string)` `html_header_splits`
embed_query using watsonx	A method used to embed a single piece of text (e.g., for the purpose of comparing it to other embedded pieces of text).	`query = "How are you?"` `query_result = watsonx_embedding.embed_query(query)`
embed_documents using watsonx	A method commonly used in various contexts for embedding documents within other documents, or in machine learning for embedding text data.	`doc_result = watsonx_embedding.embed_documents(chunks)`
TextLoader	LangChain's TextLoader is a useful tool for loading and processing text data, making it ready for use with large language models (LLMs).	`!wget "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/BYlUHaillwM8EUItaIytHQ/companypolicies.txt"`
Embedding model	Embedding models are specifically designed to interface with text embeddings.	`from ibm_watsonx_ai.metanames import EmbedTextParamsMetaNames` `from langchain_ibm import WatsonxEmbeddings` `embed_params = {`

	Embeddings generate a vector representation for a given piece of text. This is advantageous as it allows you to conceptualize text within a vector space. Consequently, you can perform operations such as semantic search, where you identify pieces of text that are most similar within the vector space.	<pre> EmbedTextParamsMetaNames.TRUNCATE_INPUT_TOKENS: 3, EmbedTextParamsMetaNames.RETURN_OPTIONS: {"input_text": True}, } watsonx_embedding = WatsonxEmbeddings(model_id="ibm/slate-125m-english-rtrvr", url="https://us-south.ml.cloud.ibm.com", project_id="skills-network", params=embed_params,)</pre>
Using Chroma DB to store embeddings	Refers to using the embedding model to create embeddings for each chunk and then storing them in the Chroma database.	<pre>vectordb = Chroma.from_documents(chunks, watsonx_embedding, ids=ids)</pre>
Similarity search	A vector database that involves finding items that are most similar to a given query item based on their vector representations. In this process, data objects are converted into vectors (which you've already done), and the search algorithm identifies and retrieves those with the closest vector distances to the query, enabling efficient and accurate identification of similar items in large datasets. Here is an example of how to perform a similarity search based on the query "Email policy."	<pre>query = "Email policy" docs = vectordb.similarity_search(query) docs</pre>

<table>
<tr>
<td>Using FIASS DB to store embeddings</td>
<td>FIASS is another vector database that is supported by LangChain.

The process of building and using FAISS is similar to Chroma DB.

However, there may be differences in the retrieval results between FAISS and Chroma DB.</td>
<td><pre><code>faissdb = FAISS.from_documents(chunks, watsonx_embedding, ids=ids)</code></pre></td>
</tr>
<tr>
<td>Defining helper functions</td>
<td>Helper functions are smaller, reusable functions that perform specific tasks and can be called within other functions to simplify code and avoid repetition. They help make code more modular, readable, and maintainable.</td>
<td><pre><code>def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
warnings.filterwarnings('ignore')</code></pre></td>
</tr>
<tr>
<td>mixtral-8x7b-instruct-v01</td>
<td>An LLM model developed by Mistral AI. It’s a Sparse Mixture of Experts (SMoE) model, which means it uses a combination of different expert models to generate high-quality text outputs.</td>
<td><pre><code>def llm():
    model_id = 'mistralai/mixtral-8x7b-instruct-v01'

    parameters = {
        GenParams.MAX_NEW_TOKENS: 256, # this controls the maximum number of tokens in the generated output
        GenParams.TEMPERATURE: 0.5, # this randomness or creativity of the model's responses
    }

    credentials = {
        "url": "https://us-south.ml.cloud.ibm.com"
    }

    project_id = "skills-network"

    model = ModelInference(</code></pre></td>
</tr>
</table>

		`model_id=model_id,` `params=parameters,` `credentials=credentials,` `project_id=project_id` `)` `mixtral_llm = WatsonxLLM(model = model)` `return mixtral_llm`
MMR retrieval	MMR in vector stores is a technique used to balance the relevance and diversity of retrieved results. It selects documents that are both highly relevant to the query and minimally similar to previously selected documents.	`retriever = vectordb.as_retriever(search_type="mmr")` `docs = retriever.invoke(query)` `docs`
Similarity score threshold retrieval	You can set a retrieval method that defines a similarity score threshold, returning only documents with a score above that threshold.	`retriever = vectordb.as_retriever(` `search_type="similarity_score_threshold",` `search_kwargs={"score_threshold": 0.4}` `)` `docs = retriever.invoke(query)` `docs`
Self-Querying Retriever	A Self-Querying Retriever has the ability to query itself. Specifically, given a natural language query, the retriever uses a query-constructing LLM chain to generate a structured query. It then applies this structured query to its underlying vector store. This enables the retriever to not only use the user-input query for semantic similarity comparison with the contents of stored documents but also to	`from langchain_core.documents import Document` `from langchain.chains.query_constructor.base import AttributeInfo` `from langchain.retrievers.self_query.base import SelfQueryRetriever` `from lark import lark`

	extract and apply filters based on the metadata of those documents.	
Parent Document Retriever	When splitting documents for retrieval, there are often conflicting desires: • You may want to have small documents so that their embeddings can most accurately reflect their meaning. If the documents are too long, the embeddings can lose meaning. • You want to have long enough documents so that the context of each chunk is retained. The Parent Document Retriever strikes that balance by splitting and storing small chunks of data.	`from langchain.retrievers import ParentDocumentRetriever` `from langchain_text_splitters import CharacterTextSplitter` `from langchain.storage import InMemoryStore`
Multi-Query Retriever	The Multi Query Retriever uses an LLM to generate multiple queries from different perspectives for a given user input query. For each query, it retrieves a set of relevant documents and then takes the unique union of these results to form a larger set of potentially relevant documents.	`def text_to_emb(list_of_text,max_input=512):` `    data_token_index  = tokenizer.batch_encode_plus(list_of_text, add_special_tokens=True,padding=True,truncation=True,max_length=max_input)` `question_embeddings=aggregate_embeddings(data_token_index['input_ids'], data_token_index['attention_mask'])` `    return question_embeddings`
sum calculator	An application that can calculate the sum of your input numbers in Gradio.	`import gradio as gr` `def add_numbers(Num1, Num2):`

<table>
<tr>
<td></td>
<td></td>
<td><pre><code>   return Num1 + Num2

Define the interface
demo = gr.Interface(
    fn=add_numbers,
    inputs=[gr.Number(), gr.Number()], # Create two
numerical input fields where users can enter numbers
    outputs=gr.Number() # Create numerical output fields
)

Launch the interface
demo.launch(server_name="127.0.0.1", server_port= 7860)</code></pre></td>
</tr>
<tr>
<td>Integrate application into Gradio</td>
<td>You can integrate an application with Gradio to leverage a web interface for inputting questions and receiving responses.

This code guides you through this integration process. It includes three components:
Initializing the model
Defining the function that generates responses from the LLM
Constructing the Gradio interface, enabling interaction with the LLM</td>
<td><pre><code># Import necessary packages
from ibm_watsonx_ai.foundation_models import
ModelInference
from ibm_watsonx_ai.metanames import
GenTextParamsMetaNames as GenParams
from ibm_watsonx_ai import Credentials
from langchain_ibm import WatsonxLLM
import gradio as gr

Model and project settings
model_id = 'mistralai/mixtral-8x7b-instruct-v01' #
Directly specifying the model

Set necessary parameters
parameters = {
    GenParams.MAX_NEW_TOKENS: 256, # Specifying the max
tokens you want to generate
    GenParams.TEMPERATURE: 0.5, # This randomness or
creativity of the model's responses
}</code></pre></td>
</tr>
</table>

<table>
<tr>
<td></td>
<td></td>
<td><pre><code>project_id = "skills-network"

Wrap up the model into WatsonxLLM inference
watsonx_llm = WatsonxLLM(
    model_id=model_id,
    url="https://us-south.ml.cloud.ibm.com",
    project_id=project_id,
    params=parameters,
)

Function to generate a response from the model
def generate_response(prompt_txt):
    generated_response = watsonx_llm.invoke(prompt_txt)
    return generated_response

Create Gradio interface
chat_application = gr.Interface(
    fn=generate_response,
    allow_flagging="never",
    inputs=gr.Textbox(label="Input", lines=2,
placeholder="Type your question here..."),
    outputs=gr.Textbox(label="Output"),
    title="Watsonx.ai Chatbot",
    description="Ask any question and the chatbot will
try to answer."
)

Launch the app
chat_application.launch(server_name="127.0.0.1",
server_port= 7860)</code></pre></td>
</tr>
<tr>
<td>Initialize the LLM</td>
<td>You can initialize the LLM by creating an instance of WatsonxLLM, a class in</td>
<td><pre><code>## LLM
def get_llm():
    model_id = 'mistralai/mixtral-8x7b-instruct-v01'</code></pre></td>
</tr>
</table>

	langchain_ibm. WatsonxLLM can use several underlying foundational models. In this snippet, you use Mixtral 8x7B. To initialize the LLM, paste the following code into qabot.py. Note that you are initializing the model with a temperature of 0.5, and allowing for the generation of a maximum of 256 tokens.	<pre> parameters = { GenParams.MAX_NEW_TOKENS: 256, GenParams.TEMPERATURE: 0.5, } project_id = "skills-network" watsonx_llm = WatsonxLLM(model_id=model_id, url="https://us-south.ml.cloud.ibm.com", project_id=project_id, params=parameters,) return watsonx_llm</pre>
Define the PDF document loader	You use the PyPDFLoader class from the langchain_community library to load PDF documents. You create the PDF loader as an instance of PyPDFLoader. Then, you load the document and return the loaded document. To incorporate the PDF loader in your bot, add the following to qabot.py:	<pre>## Document loader def document_loader(file): loader = PyPDFLoader(file.name) loaded_document = loader.load() return loaded_document</pre>
Define the text splitter	You define a document splitter that will split the text into chunks. Add the following code to aqbot.py to define such a text splitter. Note that, in this example, you are defining a RecursiveCharacterTextSplitter with a chunk size of 1000, although other splitters or parameter values are possible:	<pre>## Text splitter def text_splitter(data): text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=50, length_function=len,) chunks = text_splitter.split_documents(data) return chunks</pre>
Define the vector store	Add this code to qabot.py to define a function that embeds the chunks	<pre>## Vector db def vector_database(chunks):</pre>

	using a yet-to-be-defined embedding model and stores the embeddings in a ChromaDB vector store:	`    embedding_model = watsonx_embedding()` `    vectordb = Chroma.from_documents(chunks, embedding_model)` `    return vectordb`
Define the embedding model	Defines a watsonx_embedding() function that returns an instance of WatsonxEmbeddings, a class from langchain_ibm that generates embeddings. In this case, the embeddings are generated using IBM's Slate 125M English embeddings model. Paste this code into the qabot.py file.	`## Embedding model` `def watsonx_embedding():` `    embed_params = {` `        EmbedTextParamsMetaNames.TRUNCATE_INPUT_TOKENS: 3,` `        EmbedTextParamsMetaNames.RETURN_OPTIONS: {"input_text": True},` `    }` `    watsonx_embedding = WatsonxEmbeddings(` `        model_id="ibm/slate-125m-english-rtrvr",` `        url="https://us-south.ml.cloud.ibm.com",` `        project_id="skills-network",` `        params=embed_params,` `    )` `    return watsonx_embedding`
Define the retriever	Define a vector store-based retriever that retrieves information using a simple similarity search. To do so, add the following lines to qabot.py.	`## Retriever` `def retriever(file):` `    splits = document_loader(file)` `    chunks = text_splitter(splits)` `    vectordb = vector_database(chunks)` `    retriever = vectordb.as_retriever()` `    return retriever`
Define a question-answering chain	Use RetrievalQA from langchain, a chain that performs natural-language question-answering over a data source using retrieval-augmented generation (RAG). Add the following code to qabot.py to define a question-answering chain.	`## QA Chain` `def retriever_qa(file, query):` `    llm = get_llm()` `    retriever_obj = retriever(file)` `    qa = RetrievalQA.from_chain_type(llm=llm,` `                                    chain_type="stuff",`

		`retriever=retriever_obj,` `return_source_documents=False)` `    response = qa.invoke(query)` `    return response['result']`
Setup the Gradio interface	A Gradio interface should include: • A file upload functionality (provided by the File class in Gradio) • An input textbox where the question can be asked (provided by the Textbox class in Gradio) • An output textbox where the question can be answered (provided by the Textbox class in Gradio) Add the following code to qabot.py to add the Gradio interface.	`# Create Gradio interface` `rag_application = gr.Interface(` `    fn=retriever_qa,` `    allow_flagging="never",` `    inputs=[` `        gr.File(label="Upload PDF File", file_count="single", file_types=['.pdf'], type="filepath"),  # Drag and drop file upload` `        gr.Textbox(label="Input Query", lines=2, placeholder="Type your question here...")` `    ],` `    outputs=gr.Textbox(label="Output"),` `    title="RAG Chatbot",` `    description="Upload a PDF document and ask any question. The chatbot will try to answer using the provided document."` `)`
Add code to launch the application	Add this line to qabot.py to launch the application using port 7860.	`# Launch the app` `rag_application.launch(server_name="0.0.0.0", server_port= 7860)`
Verify	The qabot.py should look like this:	`from ibm_watsonx_ai.foundation_models import ModelInference` `from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams` `from ibm_watsonx_ai.metanames import EmbedTextParamsMetaNames`

		<pre>from ibm_watsonx_ai import Credentials from langchain_ibm import WatsonxLLM, WatsonxEmbeddings from langchain.text_splitter import RecursiveCharacterTextSplitter from langchain_community.vectorstores import Chroma from langchain_community.document_loaders import PyPDFLoader from langchain.chains import RetrievalQA import gradio as gr # You can use this section to suppress warnings generated by your code: def warn(*args, **kwargs): pass import warnings warnings.warn = warn warnings.filterwarnings('ignore') ## LLM def get_llm(): model_id = 'mistralai/mixtral-8x7b-instruct-v01' parameters = { GenParams.MAX_NEW_TOKENS: 256, GenParams.TEMPERATURE: 0.5, } project_id = "skills-network" watsonx_llm = WatsonxLLM(model_id=model_id, url="https://us-south.ml.cloud.ibm.com", project_id=project_id, params=parameters,) return watsonx_llm</pre>

		<pre>## Document loader def document_loader(file): loader = PyPDFLoader(file.name) loaded_document = loader.load() return loaded_document ## Text splitter def text_splitter(data): text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=50, length_function=len,) chunks = text_splitter.split_documents(data) return chunks ## Vector db def vector_database(chunks): embedding_model = watsonx_embedding() vectordb = Chroma.from_documents(chunks, embedding_model) return vectordb ## Embedding model def watsonx_embedding(): embed_params = { EmbedTextParamsMetaNames.TRUNCATE_INPUT_TOKENS: 3, EmbedTextParamsMetaNames.RETURN_OPTIONS: {"input_text": True}, } watsonx_embedding = WatsonxEmbeddings(model_id="ibm/slate-125m-english-rtrvr",</pre>

```
        url="https://us-south.ml.cloud.ibm.com",
        project_id="skills-network",
        params=embed_params,
    )
    return watsonx_embedding

## Retriever
def retriever(file):
    splits = document_loader(file)
    chunks = text_splitter(splits)
    vectordb = vector_database(chunks)
    retriever = vectordb.as_retriever()
    return retriever

## QA Chain
def retriever_qa(file, query):
    llm = get_llm()
    retriever_obj = retriever(file)
    qa = RetrievalQA.from_chain_type(llm=llm,
                                    chain_type="stuff",

retriever=retriever_obj,

return_source_documents=False)
    response = qa.invoke(query)
    return response['result']


# Create Gradio interface
rag_application = gr.Interface(
    fn=retriever_qa,
    allow_flagging="never",
    inputs=[
```

		`` gr.File(label="Upload PDF File", file_count="single", file_types=['.pdf'], type="filepath"), # Drag and drop file upload gr.Textbox(label="Input Query", lines=2, placeholder="Type your question here...")], outputs=gr.Textbox(label="Output"), title="RAG Chatbot", description="Upload a PDF document and ask any question. The chatbot will try to answer using the provided document.") # Launch the app rag_application.launch(server_name="0.0.0.0", server_port= 7860) ```
Serve the application	To serve the application, paste this code into your Python terminal:	`python3.11 qabot.py`