Cheat Sheet: Project: Generative AI Applications with RAG and LangChain

<table>
<tr><th>Package/Method</th><th>Description</th><th>Code example</th></tr>
<tr><td>Load method</td><td>Loads data from a server and puts the returned data into the selected element.</td><td><pre><code>data = loader.load()</code></pre></td></tr>
<tr><td>Document object</td><td>Contains information about 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.</td><td><pre><code>from langchain_core.documents import Document

Document(page_content="""Python is an interpreted high-level general-purpos
                        Python's design philosophy emphasizes code readabil
         metadata={
             'my_document_id' : 234234,
             'my_document_source' : "About Python",
             'my_document_create_time' : 1680013019
         })</code></pre></td></tr>
<tr><td>pprint function</td><td>A function in Python used to "pretty-print" data structures, making them more readable and easier to understand.</td><td><pre><code>pprint(data[0].page_content[:1000])</code></pre></td></tr>
<tr><td>PyPDFLoader</td><td>Simplifies the process of loading PDF documents into a format that can be easily manipulated and analyzed within your applications.</td><td><pre><code>pdf_url = "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.clo

loader = PyPDFLoader(pdf_url)

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/e

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

data = loader.load()

data</code></pre></td></tr>
</table>

Package/Method	Description	Code example
JSONLoader	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.	`!wget 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/h`
CSVLoader	CSV files are a common format for storing tabular data. The CSVLoader provides a convenient way to read and process this data.	`!wget 'https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/I`
UnstructuredCSVLoader	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.	`loader = UnstructuredCSVLoader(` `    file_path="mlb-teams-2012.csv", mode="elements"` `)` `data = loader.load()` `data[0].page_content` `print(data[0].metadata["text_as_html"])`
BeautifulSoup	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.	`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())`
WebBaseLoader	LangChain's tool designed to extract all text from HTML webpages and convert it into a document format suitable for further processing.	`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://ww` `data = loader.load()` `data`

Package/Method	Description	Code example
Docx2txtLoader	Utilized to convert Word documents into a document format suitable for further processing.	<pre>!wget https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/94 loader = Docx2txtLoader(“file-sample.docx”) data = loader.load() data</pre>
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.	<pre>loader = UnstructuredFileLoader("companypolicies.txt") data = loader.load() data</pre>
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.	<pre>loader = UnstructuredFileLoader("markdown-sample.md") data = loader.load() data</pre>
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.	<pre>files = ["markdown-sample.md", "companypolicies.txt"] loader = UnstructuredFileLoader(files) data = loader.load() data</pre>
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 and the specific model you are accessing.	<pre>def llm_model(model_id): parameters = { GenParams.MAX_NEW_TOKENS: 256, # this controls the maximum number GenParams.TEMPERATURE: 0.5, # this randomness or creativity of the } 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>

<table>
<tr><th>Package/Method</th><th>Description</th><th>Code example</th></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>Load source document</td><td>Loading a source document into a large language model (LLM) involves providing the model with specific data or text that it can be used to generate responses or perform tasks.</td><td><pre>!wget "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/d</pre></td></tr>
<tr><td>LangChain prompt template</td><td>A prompt template is set up using LangChain to make it reusable.</td><td><pre>template = """According to the document content here
            {content},
            answer this question
            {question}.
            Do not try to make up the answer.

        YOUR RESPONSE:

"""
prompt_template = PromptTemplate(template=template, input_variables=['conte
prompt_template</pre></td></tr>
<tr><td>Use mixtral model</td><td>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.</td><td><pre>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'])</pre></td></tr>
<tr><td>Use Llama 3 model</td><td>The Llama model (Large Language Model Meta AI) is a family of autoregressive large language models developed by Meta AI.</td><td><pre>query_chain = LLMChain(llm=llama_llm, prompt=prompt_template)
query_chain</pre></td></tr>
<tr><td>Use one piece of information</td><td>Using this code snippet, retrieve one piece of information related to the query and put it in the content variable.</td><td><pre>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 t
And my report is this: the State of the Union is strong—because you, the Am

"""</pre></td></tr>
</table>

<table>
<tr><th>Package/Method</th><th>Description</th><th>Code example</th></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>Split by Character</td><td>This is the simplest method of splitting text, which splits the text based on characters (by default "\n\n") and measures chunk length by the number of characters.</td><td><pre><code>from langchain.text_splitter import CharacterTextSplitter

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>
<tr><td>Markdown Header Text Splitter</td><td>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.</td><td><pre><code>markdown_splitter = MarkdownHeaderTextSplitter(headers_to_split_on=headers_
md_header_splits = markdown_splitter.split_text(md)
md_header_splits</code></pre></td></tr>
<tr><td>Split by HTML</td><td>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.</td><td><pre><code>html_splitter = HTMLHeaderTextSplitter(headers_to_split_on=headers_to_split
html_header_splits = html_splitter.split_text(html_string)
html_header_splits</code></pre></td></tr>
</table>

<table>
<tr><th>Package/Method</th><th>Description</th><th>Code example</th></tr>
<tr><td></td><td></td><td></td></tr>
<tr><td>embed_query using watsonx</td><td>A method used to embed a single piece of text (e.g., for the purpose of comparing it to other embedded pieces of text).</td><td><pre>query = "How are you?"

query_result = watsonx_embedding.embed_query(query)</pre></td></tr>
<tr><td>embed_documents using watsonx</td><td>A method commonly used in various contexts for embedding documents within other documents, or in machine learning for embedding text data.</td><td><pre>doc_result = watsonx_embedding.embed_documents(chunks)</pre></td></tr>
<tr><td>TextLoader</td><td>LangChain's TextLoader is a useful tool for loading and processing text data, making it ready for use with large language models (LLMs).</td><td><pre>!wget "https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/B</pre></td></tr>
<tr><td>Embedding model</td><td>Embedding models are specifically designed to interface with text embeddings.

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.</td><td><pre>from ibm_watsonx_ai.metanames import EmbedTextParamsMetaNames
from langchain_ibm import WatsonxEmbeddings

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,
)</pre></td></tr>
<tr><td>Using Chroma DB to store embeddings</td><td>Refers to using the embedding model to create embeddings for each chunk and then storing them in the Chroma database.</td><td><pre>vectordb = Chroma.from_documents(chunks, watsonx_embedding, ids=ids)</pre></td></tr>
<tr><td>Similarity search</td><td>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</td><td><pre>query = "Email policy"
docs = vectordb.similarity_search(query)
docs</pre></td></tr>
</table>

Package/Method	Description	Code example
	identification of similar items in large datasets. Here is an example of how to perform a similarity search based on the query “Email policy.”	
Using FAISS DB to store embeddings	FAISS 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.	`faissdb = FAISS.from_documents(chunks, watsonx_embedding, ids=ids)`
Defining helper functions	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.	`def warn(*args, **kwargs):` `    pass` `import warnings` `warnings.warn = warn` `warnings.filterwarnings('ignore')`
mixtral-8x7b-instruct-v01	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.	`def llm():` `    model_id = 'mistralai/mixtral-8x7b-instruct-v01'` `parameters = {` `    GenParams.MAX_NEW_TOKENS: 256,  # this controls the maximum number of t` `    GenParams.TEMPERATURE: 0.5, # this randomness or creativity of the mode` `}` `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` `)` `mixtral_llm = WatsonxLLM(model = model)` `return mixtral_llm`

<table>
<tr><th>Package/Method</th><th>Description</th><th>Code example</th></tr>
<tr><td>MMR retrieval</td><td>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.</td><td>

```
retriever = vectordb.as_retriever(search_type="mmr")
docs = retriever.invoke(query)
docs
```

</td></tr>
<tr><td>Similarity score threshold retrieval</td><td>You can set a retrieval method that defines a similarity score threshold, returning only documents with a score above that threshold.</td><td>

```
dretriever = vectordb.as_retriever(
    search_type="similarity_score_threshold", search_kwargs={"score_thresho
)
docs = retriever.invoke(query)
docs
```

</td></tr>
<tr><td>Self-Querying Retriever</td><td>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 extract and apply filters based on the metadata of those documents.</td><td>

```
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
```

</td></tr>
<tr><td>Parent Document Retriever</td><td>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.</td><td>

```
from langchain.retrievers import ParentDocumentRetriever
from langchain_text_splitters import CharacterTextSplitter
from langchain.storage import InMemoryStore
```

</td></tr>
<tr><td>Multi-Query Retriever</td><td>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.</td><td>

```
def text_to_emb(list_of_text,max_input=512):
    data_token_index  = tokenizer.batch_encode_plus(list_of_text, add_speci
    question_embeddings=aggregate_embeddings(data_token_index['input_ids'],
    return question_embeddings
```

</td></tr>
<tr><td>sum calculator</td><td>An application that can calculate the sum of your input numbers in Gradio.</td><td>

```
import gradio as gr


def add_numbers(Num1, Num2):
    return Num1 + Num2
```

</td></tr>
</table>

Package/Method	Description	Code example
		`` # Define the interface demo = gr.Interface(fn=add_numbers, inputs=[gr.Number(), gr.Number()], # Create two numerical input fields outputs=gr.Number() # Create numerical output fields) # Launch the interface demo.launch(server_name="127.0.0.1", server_port= 7860) ```
Integrate application into Gradio	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	``` # Import necessary packages from ibm_watsonx_ai.foundation_models import ModelInference from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as Gen 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 # Set necessary parameters parameters = { GenParams.MAX_NEW_TOKENS: 256, # Specifying the max tokens you want to GenParams.TEMPERATURE: 0.5, # This randomness or creativity of the mode } 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", ```

Package/Method	Description	Code example
		inputs=gr.Textbox(label="Input", lines=2, placeholder="Type your questi 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) </td>
Initialize the LLM	You can initialize the LLM by creating an instance of WatsonxLLM, a class in 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.	## 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
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 code to qabot.py.	## Document loader def document_loader(file): loader = PyPDFLoader(file.name) loaded_document = loader.load() return loaded_document
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.	## 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
Define the vector store	Add this code to qabot.py to define a function that embeds the chunks using a yet-to-be-defined embedding model and	## Vector db def vector_database(chunks): embedding_model = watsonx_embedding() vectordb = Chroma.from_documents(chunks, embedding_model) return vectordb

<table>
<tr><th>Package/Method</th><th>Description</th><th>Code example</th></tr>
<tr><td></td><td>stores the embeddings in a ChromaDB vector store.</td><td></td></tr>
<tr><td>Define the embedding model</td><td>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.</td><td><pre><code>## 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</code></pre></td></tr>
<tr><td>Define a question-answering chain</td><td>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.</td><td><pre><code>## 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']</code></pre></td></tr>
<tr><td>Setup the Gradio interface</td><td>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.</td><td><pre><code># 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=['
        gr.Textbox(label="Input Query", lines=2, placeholder="Type your que
    ],
    outputs=gr.Textbox(label="Output"),
    title="RAG Chatbot",
    description="Upload a PDF document and ask any question. The chatbot wi
)</code></pre></td></tr>
<tr><td>Add code to launch the application</td><td>Add this line to qabot.py to launch the application using port 7860.</td><td><pre><code># Launch the app
rag_application.launch(server_name="0.0.0.0", server_port= 7860)</code></pre></td></tr>
<tr><td>Verify</td><td>The qabot.py should look like this.</td><td><pre><code>from ibm_watsonx_ai.foundation_models import ModelInference
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams
from ibm_watsonx_ai.metanames import EmbedTextParamsMetaNames
from ibm_watsonx_ai import Credentials
from langchain_ibm import WatsonxLLM, WatsonxEmbeddings</code></pre></td></tr>
</table>

Package/Method	Description	Code example

```
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

## 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},
    }
```

Package/Method	Description	Code example
		<pre> 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 ## 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=[' gr.Textbox(label="Input Query", lines=2, placeholder="Type your que], outputs=gr.Textbox(label="Output"), title="RAG Chatbot", description="Upload a PDF document and ask any question. The chatbot wi) # Launch the app rag_application.launch(server_name="0.0.0.0", server_port= 7860)</pre>
Package/Method	Description	
Serve the application	To serve the application, paste this code into your Python terminal:	`python3.11 qabot.py`

Skills Network