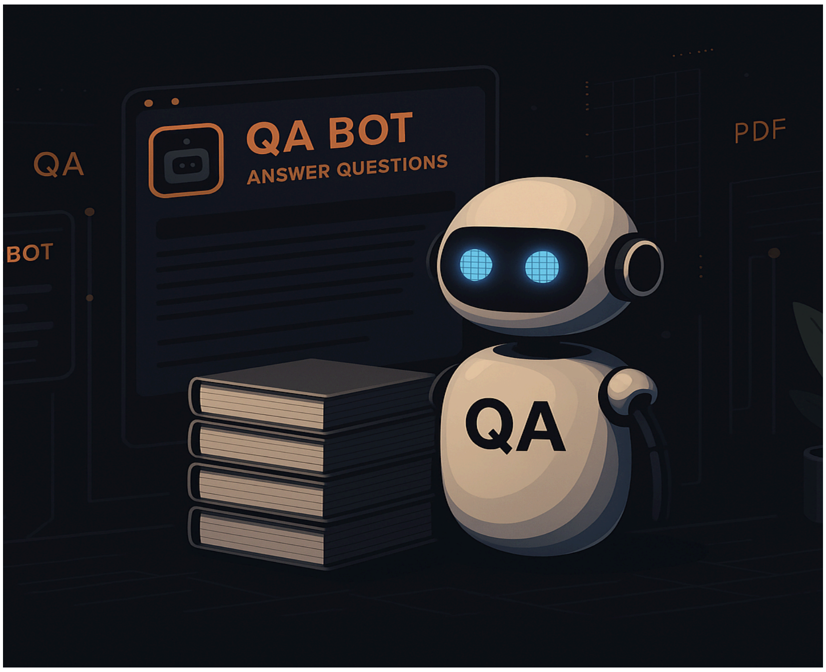
I Built my very first AI agent (QA Bot That Leverages LangChain and LLMs to Answer Questions from Loaded Documents)

In this final wrap-up project, I brought together all the skills I've gained throughout my learning journey to build a fully functional question-answering (QA) bot. I used LangChain and a large language model (LLM) to create a system that can accurately answer questions based on the content of loaded PDF documents.

I integrated several components into this project: document loaders, text splitters, embedding models, vector databases, retrievers, and a Gradio interface for the front end. The result is an intelligent assistant capable of quickly and efficiently responding to queries based on a company's extensive library of documents, ranging from legal contracts to technical manuals, without the need for manual searching.

The bot I developed automates the process of reading and understanding complex PDF content. By combining the power of LangChain and an LLM, it delivers precise and contextually accurate answers to user queries. I designed the system to be both powerful and user-friendly, wrapping all the backend logic into a seamless interface using Gradio.



Source: DALL-E

What I Learned

Through this project, I learned how to:

- Integrate document loaders, text splitters, embedding models, and vector databases into a cohesive pipeline
- Utilize LangChain and LLMs to retrieve and answer questions from large PDF documents
- Design and build an end-to-end QA system with a user-friendly Gradio interface

Setting Up My Project Environment

To kick off the development of my QA bot, I began by setting up a Python virtual environment. Using a virtual environment helps me manage project-specific dependencies without conflicts between package versions across different projects.

Inside the terminal of my Cloud IDE, I navigated to the /home/project directory and ran the following commands:

```
pip install virtualenv
virtualenv my_env # create a virtual environment
named my_env source my_env/bin/activate #
activate my_env
```

This setup ensured I had a clean and isolated workspace to build and test my QA system without interference from other Python packages.

Installing the Necessary Libraries

To ensure that my QA bot runs smoothly, I needed to install several prerequisite libraries. Some functions in my scripts rely on external packages, so I took the time to set everything up properly from the start.

For this project, I used:

- **Gradio** to quickly build a user-friendly web interface for interacting with my AI bot
- **IBM watsonx AI** to access powerful LLMs through IBM's watsonx.ai API
- **LangChain**, along with langchain-ibm and langchain-community, to tie all the components together and handle retrieval-augmented generation
- **ChromaDB** as the vector database to store and retrieve document embeddings
- **PyPDF** for parsing and loading PDF documents
- **Pydantic** to handle data validation and configuration settings

To install everything inside my virtual environment (my_env), I ran the following command in the terminal:

Here's how to install these packages (from your terminal):

```
# installing necessary pacakges
in my_env python3.11 -m pip
install \ gradio==4.44.0 \
ibm-watsonx-ai==1.1.2
\ langchain==0.2.11 \
langchain-community=
=0.2.10 \
langchain-ibm==0.1.11
\ chromadb==0.4.24 \
pypdf==4.3.1 \
pydantic==2.9.1
```

Constructing My QA Bot

With my environment ready, I moved on to the exciting part, building the QA bot itself!

I started by creating a new Python file named qabot.py, which would hold all the logic for the bot. I made sure to save it correctly under that filename to keep things organized.

Importing the Necessary Libraries

Inside qabot.py, I began by importing all the required modules and classes from gradio, ibm_watsonx_ai, langchain_ibm, langchain, and langchain_community. These imports were essential for setting up the LLM, embedding models, text splitting, PDF loading, vector storage, and building the question-answering pipeline.

Here's what I added at the top of the file:

```
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
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')
```

Code breakdown

Initializing the LLM

Next, I initialized the large language model (LLM) using the WatsonxLLM class from langchain_ibm. For this project, I chose to use the **Mixtral 8x7B** model, although IBM's watsonx.ai platform supports several other powerful models, like **Llama 3.1 405B**.

To get started, I added the following snippet to my qabot.py file. I configured the model with a **temperature of 0.5** for balanced creativity and precision, and I set the **maximum token limit to 256** to control the length of the output:

```
## 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
```

Defining the PDF Document Loader

To feed content into my QA bot, I needed to load PDF documents first. I used the PyPDFLoader class from the langchain_community library for this. It made loading PDFs simple and effective.

I defined a function in qabot.py that initializes the loader with the path to the PDF, loads it, and returns the content. Here's what I added:

```
## Document loader

def document_loader(file):

  loader = PyPDFLoader(file.name)

  loaded_document = loader.load()

  return loaded_document
```

This function allowed me to take any PDF file and convert its contents into a format I could then split, embed, and search through, forming the entry point for the entire QA pipeline.

Defining the Text Splitter

Once I had the LLM set up, I needed a way to break down the loaded PDF documents into manageable chunks. The .load() method from the PDF loader brings in the full document but doesn't handle any chunking, so I had to define a text splitter manually.

To do this, I used the RecursiveCharacterTextSplitter from LangChain. It's flexible and efficient, especially for handling long documents. I chose a **chunk size of 1000 characters**, which provided a good balance between context and performance. This splitter ensures that my documents are broken down into chunks that the LLM can handle easily, enabling accurate and coherent responses during retrieval and generation. Here's the code I added to my qabot.py:

```
## 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
```

Defining the Vector Store

With the text now split into manageable chunks, I needed a way to convert those chunks into vector embeddings and store them for efficient retrieval later. To handle this, I defined a function that takes the split documents, embeds them using an embedding model (which I would define shortly), and stores the resulting vectors in a **ChromaDB** vector store.

Here's the code I added to qabot.py:

```
## Vector db

def vector_database(chunks):

  embedding_model = watsonx_embedding()

  vectordb = Chroma.from_documents(chunks, embedding_model)

  return vectordb
```

Defining the Embedding Model

Since my create_vector_store() function relied on an embedding model to convert text chunks into vector representations, the next step was to define that model. I created a function called watsonx_embedding() that returns an instance of WatsonxEmbeddings from langchain_ibm.

For this project, I used **IBM's Slate 125M English embeddings model**, which worked well for transforming text into dense vectors suitable for semantic search.

Here's the code I added to qabot.py:

```
## 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
```

A Quick Note on Function Order

It's worth mentioning that even though I defined watsonx_embedding() **after** the create_vector_store() function, Python didn't mind at all. In Python, the **order of function definitions doesn't affect functionality**, as long as the functions are called after they're defined during execution.

So technically, I could have defined watsonx_embedding() first and create_vector_store() later, and everything would still work exactly the same. It's one of those nice flexibilities Python gives us during development.

Define the retriever

I needed to define a retriever to pull chunks of the document from the vector store I had already set up. In this case, I opted for a vector store-based retriever that uses a simple similarity search to retrieve relevant information. To make this happen, I added 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

With my vector store and retriever in place, I was ready to define a question-answering chain to complete my system. For this project, I chose to use RetrievalQA from langchain, a chain that leverages retrieval-augmented generation (RAG) to perform natural-language question-answering over my data source. To implement this, I added the following code to qabot.py:

```
## 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']
```

Recap of the QA Bot's Linked Elements

I designed my QA bot by carefully linking several components to create a cohesive system. The core of the bot is the RetrievalQA chain, which I configured to accept my language model from (get_llm()) and a retriever object (generated by retriever()) as arguments. The retriever, which I built to fetch relevant document chunks, relies on the vector store I created with vector_database(). This vector store, in turn, required an embeddings model (from watsonx_embedding()) to convert text into vector representations and document chunks produced by my text splitter (from text_splitter()). To generate these chunks, I used the text splitter on raw text, which I extracted from a PDF using PyPDFLoader. By connecting these elements, I effectively defined the core functionality of my QA bot, enabling it to answer questions accurately using retrieval-augmented generation.

Set up the Gradio interface

Given that the core functionality of the bot has been created, the final item I defined was the Gradio interface. My Gradio interface includes :

- 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)

I added 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."

)
```

After that, to wrap it all up and get the QA bot to work, I added one more 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)
```

After adding the above line, I just saved qabot.py.

Here's all of the code in one snippet for clarity and ease of use!

```
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
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
```

```python
## 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",
     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, I type the following into the Python terminal:

```
python3.11 qabot.py
```

A Quick Tip on Using the Terminal (for viewers)

If you cannot find an open Python terminal or the buttons on the above cell do not work, you can launch a terminal by going to `Terminal --> New Terminal`. However, if you launch a new terminal, do not forget to source the virtual environment you created at the beginning of your tutorial before running the above line:

```
source my_env/bin/activate # activate my_env
```

🚀 Launching My Application

Now that I've built the bot, I'm ready to launch the application! If I'm working within IBM Skills Network, I can simply click the **Launch Application** button.

If that doesn't work, here's exactly what I do:

1. I open the **Skills Network extension**.
2. I click on **Launch Application**.
3. I enter the port number **(7860),** this is the server port I specified in `qabot.py`.
4. I click on **Your application** to start the bot.

If that doesn't work for some reason, I click the **Open in new browser tab** icon to manually launch it.

Once it's up and running, I can interact with the bot by uploading a readable PDF document and asking it questions about the content!

📌 **Heads up:** For best results, I make sure not to upload large PDFs, the current setup doesn't handle big files well and may fail.

When I'm done experimenting with the app, I press **Ctrl + C** in the terminal to stop the server and then close the application tab.

Sample document

- **Link to sample document:**
    - [Sample document pdf for Testing]
- **Instructions for use:**
    - Download and use this sample document for testing the QA bot.

Author(s)

Kang Wang

Wojciech “Victor” Fulmyk

Kunal Makwana

Other Contributor(s)

Hailey Quach

Skills Network IBM

How I Launch the App as a Client-Ready Gradio Web App

🌍 Launch on Hugging Face Spaces (Permanent Hosting)

Deploy your Gradio app to Hugging Face for free with GPU support and a permanent public link:

1. Install Hugging Face CLI

```
pip install huggingface_hub
```

2. Log In to Hugging Face

```
huggingface-cli login
```

Paste your access token from: https://huggingface.co/settings/tokens

3. Deploy

```
gradio deploy
```

Follow the prompts to:

- Name your Space

- Set it as public

- Choose a license (e.g., MIT)

✅ This will automatically upload your app to https://huggingface.co/spaces

✅ You'll receive a shareable link like:

```
https://huggingface.co/spaces/YourUsername/pdf-qa-agent
```