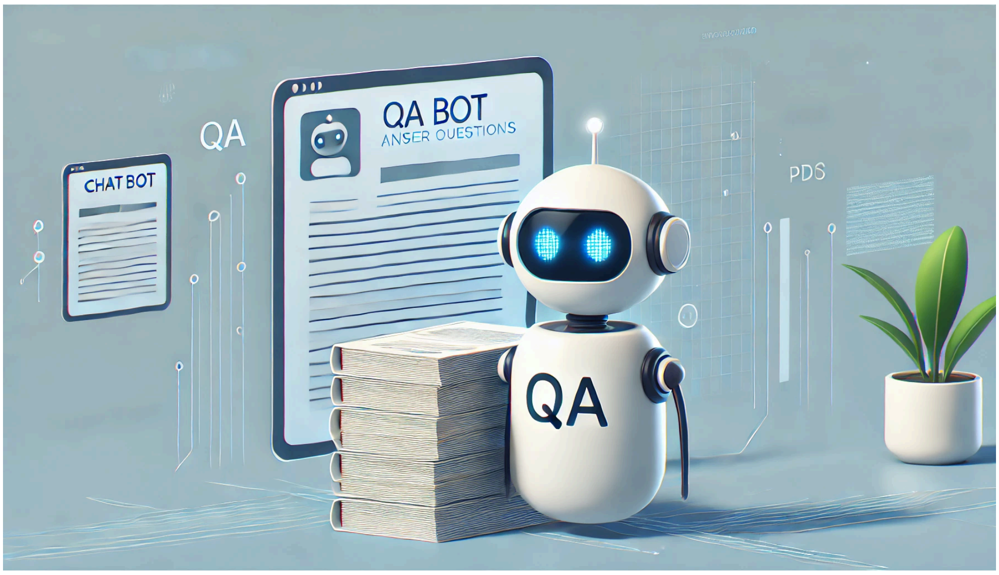
Skills Network

# Construct a QA Bot that Leverages LangChain and LLMs to Answer Questions from Loaded Documents

Estimated time needed: 60 minutes

In this project, you will construct a question-answering (QA) bot. This bot will leverage LangChain and a large language model (LLM) to answer questions based on content from loaded PDF documents. To build a fully functional QA system, you'll combine various components, including document loaders, text splitters, embedding models, vector databases, retrievers, and Gradio as the front-end interface.

Imagine you're tasked with creating an intelligent assistant that can quickly and accurately respond to queries based on a company's extensive library of PDF documents. This could be anything from legal documents to technical manuals. Manually searching through these documents would be time-consuming and inefficient.



Source: DALL-E

In this project, you will construct a QA bot that automates this process. By leveraging LangChain and an LLM, the bot will read and understand the content of loaded PDF documents, enabling it to provide precise answers to user queries. You will integrate the tools and techniques, from document loading, text splitting, embedding, vector storage, and retrieval, to create a seamless and user-friendly experience via a Gradio interface.

## Learning objectives

By the end of this project, you will be able to:

- Combine multiple components, such as document loaders, text splitters, embedding models, and vector databases, to construct a fully functional QA bot
- Leverage LangChain and LLMs to solve the problem of retrieving and answering questions based on content from large PDF documents

# Set up

## Setting up a virtual environment

Let's create a virtual environment. Using a virtual environment allows you to manage dependencies for different projects separately, avoiding conflicts between package versions.

In the terminal of your Cloud IDE, ensure that you are in the path `/home/project`, then run the following commands to create a Python virtual environment.

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

## Installing necessary libraries

To ensure seamless execution of your scripts, and considering that certain functions within these scripts rely on external libraries, it's essential to install some prerequisite libraries before you begin. For this project, the key libraries you'll need are `Gradio` for creating user-friendly web interfaces and `IBM-watsonx-AI` for leveraging advanced LLM models from the IBM watsonx API.

- `gradio` allows you to build interactive web applications quickly, making your AI models accessible to users with ease.
- `ibm-watsonx-ai` for using LLMs from IBM watsonx.ai.
- `langchain, langchain-ibm, langchain-community` for using relevant features from Langchain.
- `chromadb` for using the chroma database as a vector database.
- `pypdf` is required for loading PDF documents.

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

```
# installing necessary packages 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
```

Now, the environment is ready to create the application.

# Construct the QA bot

It's time to construct the QA bot!

Let's start by creating a new Python file to store your bot. Click the button below to create a new Python file, and call it `qabot.py`. If, for whatever reason, the button does not work, make the new file by going to `File --> New Text File`. Be sure to save the file as `qabot.py`.

Open **qabot.py** in IDE

You will populate `qabot.py` in the following sections with your bot.

## Import the necessary libraries

Inside `qabot.py`, import the following from `gradio`, `ibm_watsonx.ai`, `langchain_ibm`, `langchain`, and `langchain_community`. The imported classes are necessary for initializing models with the correct credentials, splitting text, initializing a vector store, loading PDFs, generating a question-answer retriever, and using Gradio.

```
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')
```

## Initialize the LLM

You will now initialize the LLM by creating an instance of `WatsonxLLM`, a class in `langchain_ibm`. `WatsonxLLM` can use several underlying foundational models. In this particular example, you will use Mixtral 8x7B, although you could have used other models, such as Llama 3.3 70B. For a list of foundational models available in watsonx.ai, refer to the documentation.

To initialize the LLM, paste the following 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

Next, you will define the PDF document loader. To load PDF documents, you will use the `PyPDFLoader` class from the `langchain_community` library. The syntax is quite straightforward. First, 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`:

```
## Document loader
def document_loader(file):
    loader = PyPDFLoader(file.name)
    loaded_document = loader.load()
    return loaded_document
```

## Define the text splitter

The PDF document loader loads the document but does not split it into chunks when using the `.load()` method. Consequently, you must define a document splitter that will split the text into chunks. Add the following code to `qabot.py` to define such a text splitter. 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

Now that you can load the PDF into text and split that text into chunks, you must define a way to embed and store those chunks in a vector database. Add the following code to `qabot.py` to define a function that embeds the chunks using a yet to be defined embedding model and stores the embeddings in a ChromaDB vector store:

```
## Vector db
def vector_database(chunks):
    embedding_model = watsonx_embedding()
    vectordb = Chroma.from_documents(chunks, embedding_model)
    return vectordb
```

## Define the embedding model

The above `vector_database()` function assumes the existence of a `watsonx_embedding()` function that loads an instance of an embedding model. This embedding model is needed to convert chunks of text into vector representations. The following code defines a `watsonx_embedding()` function that returns an instance of `WatsonxEmbeddings`, a class from `langchain_ibm` that generates embeddings. In this particular 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
```

It does not matter to Python that `watsonx_embedding()` is defined after `vector_database()`. The order of these definitions could have been reversed, which would have resulted in no change in the bot's underlying functionality.

## Define the retriever

Now that your vector store is defined, you must define a retriever that retrieves chunks of the document from it. In this particular case, you will 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

Finally, it is time to define a question-answering chain! In this particular example, you will 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']
```

Let's recap how all the elements in our bot are linked. Note that `RetrievalQA` accepts an LLM (`get_llm()`) and a retriever object (an instance generated by `retriever()`) as arguments. However, the retriever is based on the vector store (`vector_database()`), which in turn needed an embeddings model (`watsonx_embedding()`) and chunks generated using a text splitter (`text_splitter()`). The text splitter, in turn, needed raw text, and this text was loaded from a PDF using `PyPDFLoader`. This effectively defines the core functionality of your QA bot!

## Set up the Gradio interface

Given that you have created the core functionality of the bot, the final item to define is the Gradio interface. Your 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

Finally, you need to add one more line to `qabot.py` to launch your application using port 7860:

```
# Launch the app
rag_application.launch(server_name="0.0.0.0", server_port= 7860)
```

After adding the above line, save `qabot.py`.

## Verify `qabot.py`

Your `qabot.py` should now look like the following:

```
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
## 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, paste the following into your Python terminal:

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

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 this lab before running this line:

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

# Launch the application

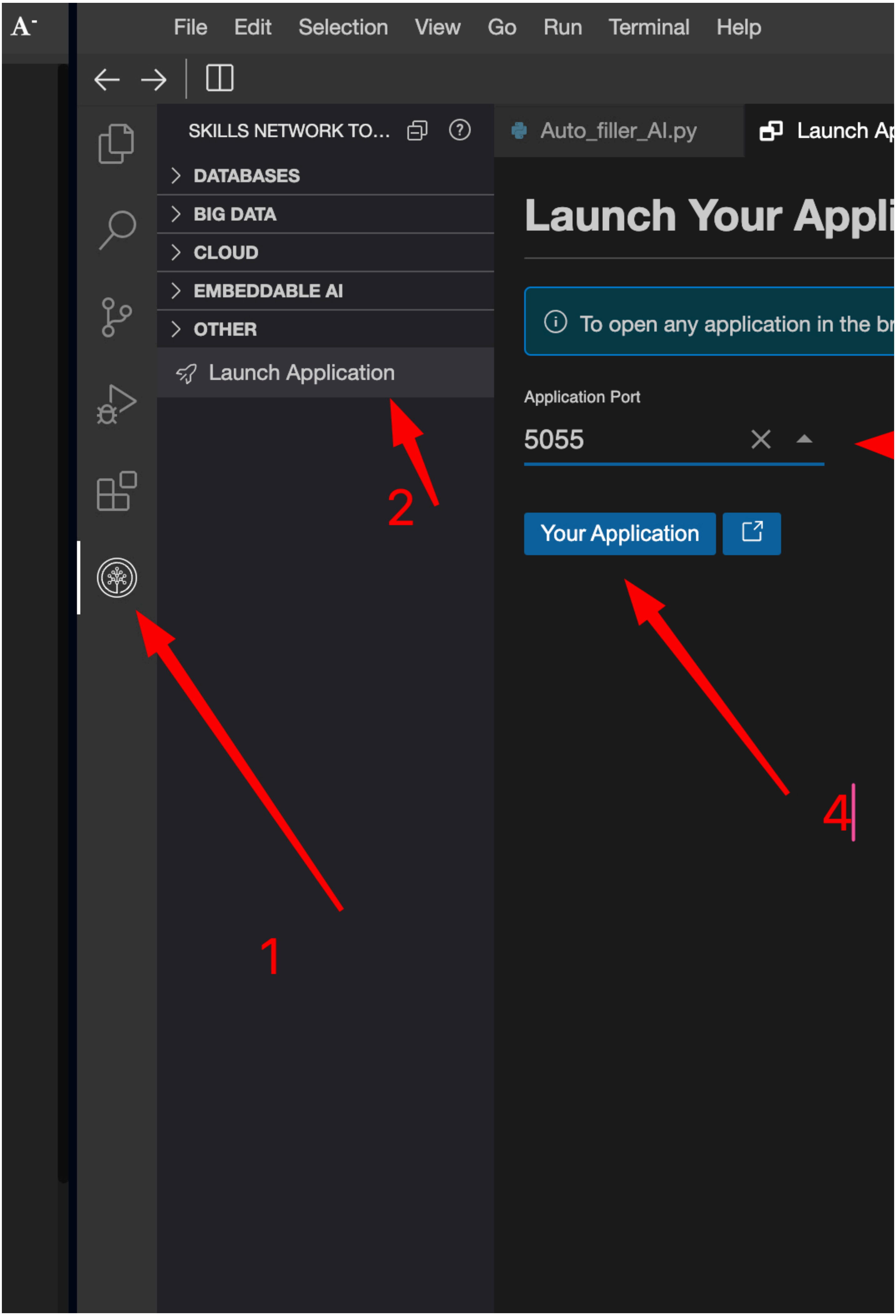
You are now ready to launch the served application! To do so, click on the following button:

Launch Application

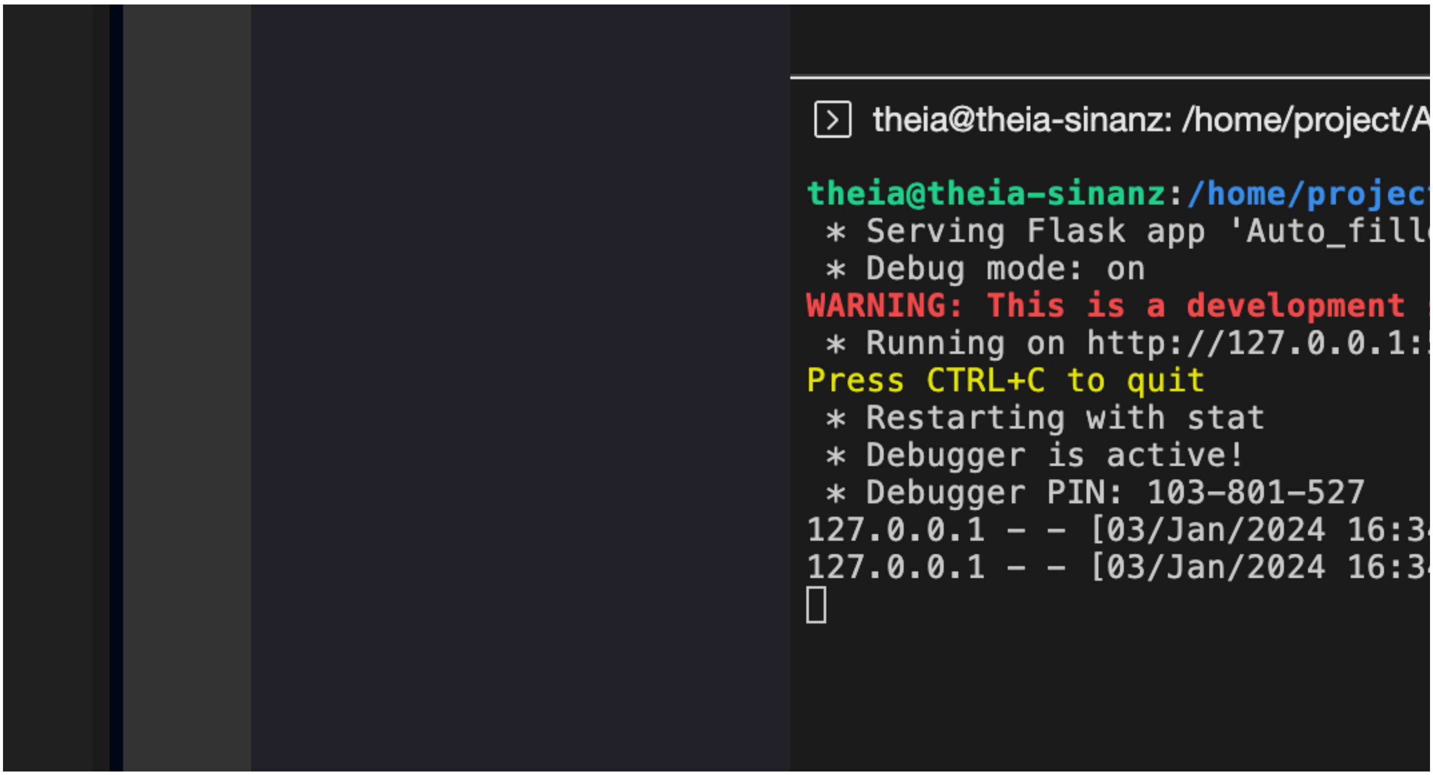
If the above button does not work, use the following instructions:

1. Select the Skills Network extension.
2. Click **Launch Application**
3. Insert the port number (in this case, 7860, which is the server port we put in `qabot.py`)
4. Click **Your application** to launch the application.

Note: If the application does not work using **Your Application**, use the icon **Open in new browser tab**.



File
Edit
Selection
View
Go
Run
Terminal
Help
SKILLS NETWORK TO...
DATABASES
BIG DATA
CLOUD
EMBEDDABLE AI
OTHER
Launch Application
Auto_filler_AI.py
Launch Ap
Launch Your Appli
To open any application in the br
Application Port
5055
Your Application
1
2
4



```
theia@theia-sinanz: /home/project/A

theia@theia-sinanz:/home/projec
 * Serving Flask app 'Auto_fill
 * Debug mode: on
WARNING: This is a development 
 * Running on http://127.0.0.1:
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 103-801-527
127.0.0.1 - - [03/Jan/2024 16:3
127.0.0.1 - - [03/Jan/2024 16:3
```

You can now interact with the application by uploading a readable PDF document and asking a question about its contents!

For best results, ensure that the PDF document is not too large. Large documents will fail with the current setup.

If you finish experimenting with the app and want to exit, press ctrl+c in the terminal and close the application tab.

# Author

## Author(s)

Kang Wang

Wojciech "Victor" Fulmyk

Kunal Makwana

Ricky Shi

## Contributor(s)

Hailey Quach