Create your own chatbot with Azure

Roland Fontanes

Table of contents

Introduction

Nowadays, with the democratization of Artificial Intelligence, chatbots are everywhere. You might already use them at work for coding or specific templates, or in your personal life for recipes or some decoration ideas. I am fond of Data and AI, so I always wanted to try creating one, but never had the chance to do so. But recently I got a really interesting proposition for an internship where my job would be to optimize their internal chatbot. So let's try to make one ourselves to see what we are capable of.

What is a chatbot?

I already knew what was a chatbot before, but let's search on Google: "What is a chatbot?". For IBM (https://www.ibm.com/think/topics/chatbots) : "A chatbot is a computer program that simulates human conversation with an end user. Not all chatbots are equipped with artificial intelligence (AI), but modern chatbots increasingly use conversational AI techniques such as natural language processing (NLP) to understand user questions and automate responses to them".

This is a long and great definition so let's try to understand each part.

First, we will code something that interact with the user and answer and/or discuss with it.

Then, AI is not mandatory, for example you can have a code like this one : https://p5js.org/tutorials/criticalai4-no-ai-chatbot. Every question and answer is already typed in the code and the chatbot only displays the corresponding text. But we don't want that; we want something that really create a new answer each time and which can answer to almost an infinite quantity of questions.

Last but not least, here we won't need to dive into Natural Language Processing (NLP), though it is a very important part. How does my computer understand what I am asking him? Does it translate my language into a programming language? Into assembly? Well, that's where NLP comes in, it allows computers to process and interpret human language in a way they can use to generate relevant answers. First it breaks your text into small parts (words, subwords, etc.) called tokens. These tokens are converted into numerical vectors, the closer the words, the closer their vectors. Then Large Language Models (LLMs) such as GPT (Generative Pre-trained Transformer) analyze these vectors to understand the context, relationships between words, and your intent. They can then generate a coherent, human-like response based on what they've learned from billions of examples during training.

What method?

Now that we know what a chatbot is, let's create one. So, like before, let's search on the Internet: "How to create a chatbot?".

First, I found Rule-based Chatbots, the simplest ones. These are the ones we discussed before, with no AI, and only predefined rules such as: if the user asks "how are you" -> answer "I'm

fine thank you, what about you?". This is too simple and cannot handle a wide range of questions.

Then, we have Retrieval-based chatbots, here, they do not rely on predefined rules, we also retrieve information from a database or documents to answer the question. The most popular one is Retrieval-Augmented Generation (RAG): you ask the question, the chatbot uses embeddings to find relevant parts of your documents, it sends the question and your relevant parts to an LLM which then generates a great answer with enough context and/or references.

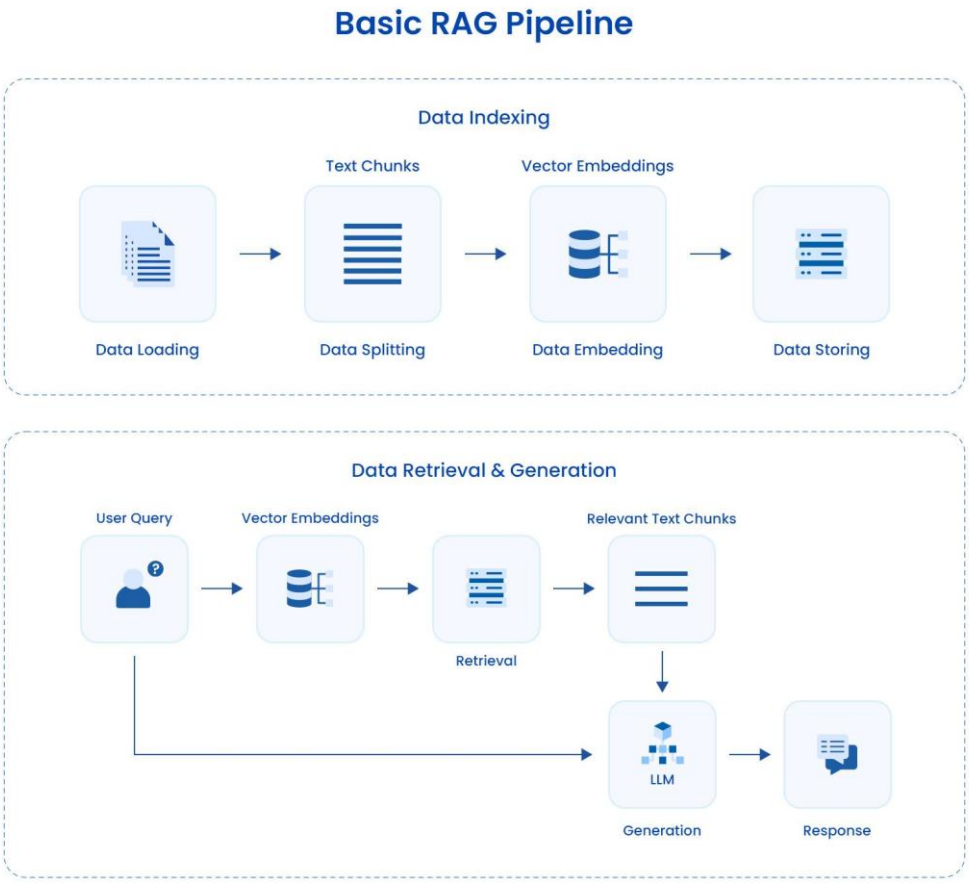

Figure 1 : *Comment un pipeline RAG transforme vos données en découvertes*, https://www.astera.com/fr/type/blog/rag-pipeline, 2025

Finally, Generative Chatbots like GPT, Gemini, or Claude (you may have heard of these names) do not retrieve data but generate an answer with what they learned in the training. They have already been given a ton of information and don't need documents to answer you. But they are much more difficult to create for an individual as you need a ton of calculus power and information.

In this project, we need a natural and efficient chatbot, without a complex installation. This is why I chose the RAG pipeline. If you understood well to that point, you need data (.json,

documents, .pdf, .txt, etc…) that you can get on your own on the internet but also a working LLM, fortunately, Azure is here for us.

Azure AI Foundry

Azure is a Microsoft cloud-based platform that allows users to create, deploy and manage computing services (SaaS, IaaS & PaaS). These services are varied, including virtual machines, databases, web apps, and more. It also provides Machine Learning and AI solutions with Azure AI Foundry.

Azure AI Foundry, includes Azure OpenAI service which gives the user access to many models such as GPT-4 and text-embedding-3-large which we will use in this project.

To create your two models you simply sign up for Azure AI Foundry (https://ai.azure.com) , and then you need a subscription and, if you are a student, this will be free of charge. Then you click on "Create new" and you will end up on this small page (Figure 2). By clicking on next you will be on this second page (Figure 3) and here you will have to choose the parameters of your project, I have no recommendation, except for the Region part. Some Region may not be compatible and Azure will give you an error if you try to create your project. If you encounter this issue, make sure to select East US, East US 2, West US or even Sweden Central.

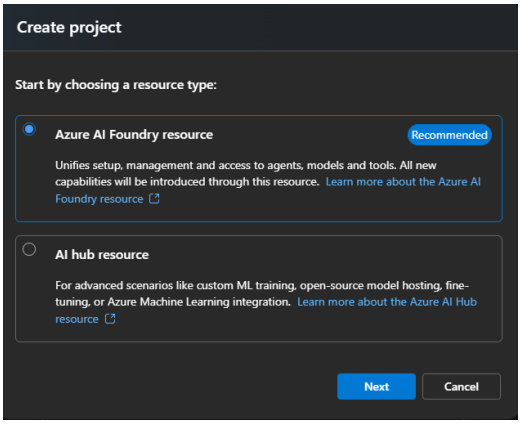

Figure 2

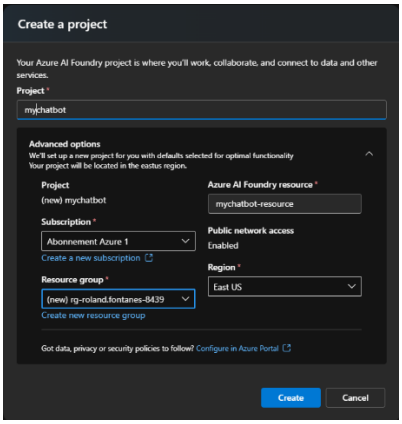

Figure 3

You will end up on this page :

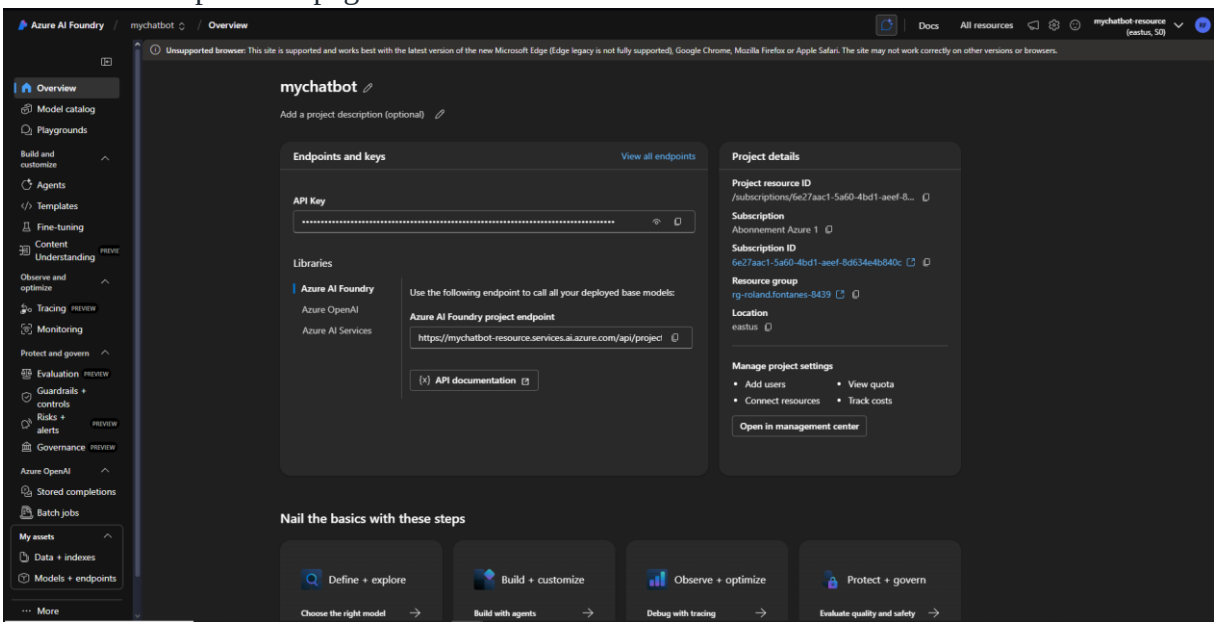

Figure 4

Make sure to note your API Key and Azure AI Foundry project endpoint because we will need them afterwards.

Then, click on Model catalog in the sidebar and in the search bar at the bottom, type ‘gpt-4o-mini’ and select the corresponding model. You are now here:

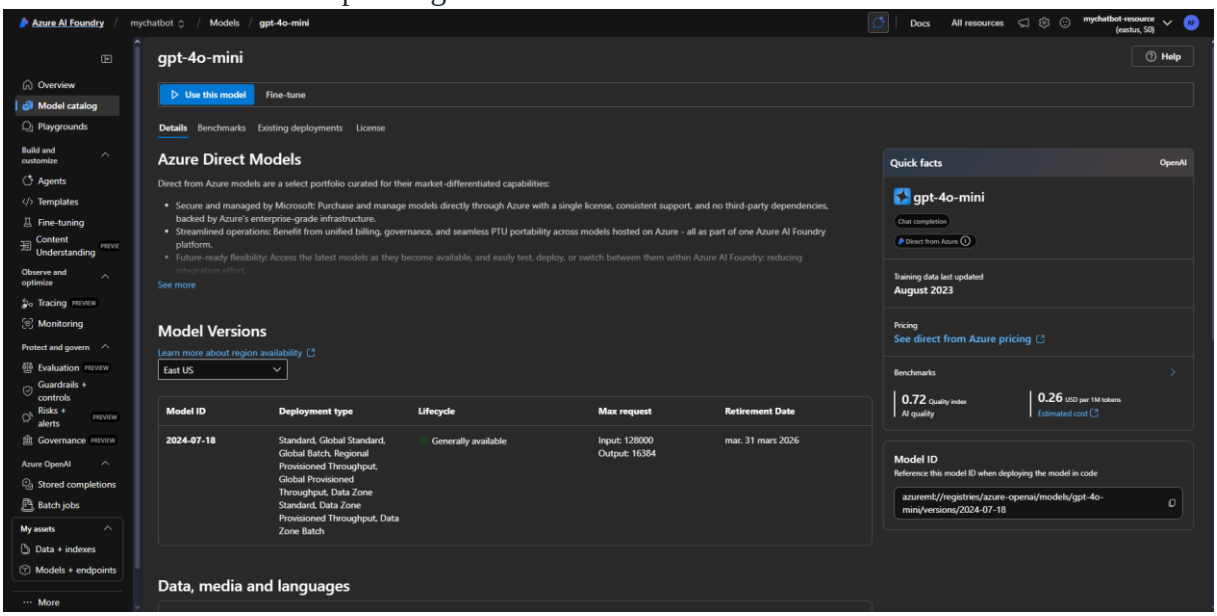

Figure 5

Click on “Use this model” and then “Deploy”. Now redo the same steps from “Model Catalog” but for the model called “text-embedding-3-large”. So now in the sidebar, in

“Models + endpoints” you should have this:

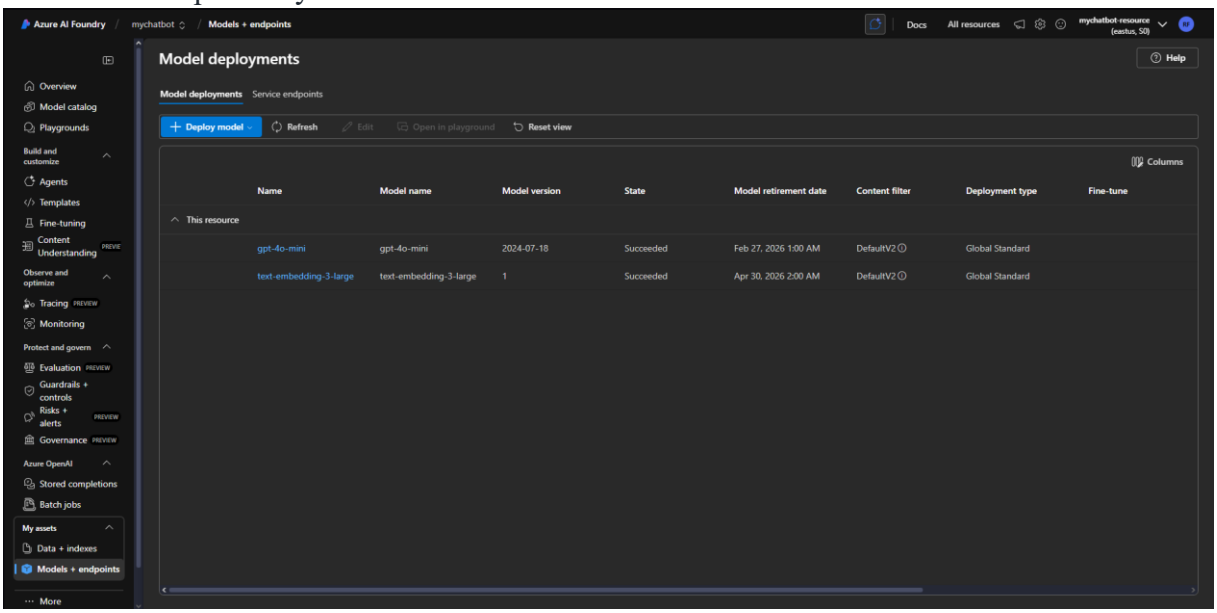

Figure 6

Python Configuration:

The Python app is divided into two parts: the RAG system and the Streamlit part.

If you are in a rush, the first thing you are going to do is copy and paste your API key and your API endpoint that we saved earlier into the .env file:

```
PROVIDER="azure"

AZURE_OPENAI_ENDPOINT="COPY/PASTE ENDPOINT"
AZURE_OPENAI_API_VERSION="2024-08-01-preview" # you can also change the openai API version here
AZURE_OPENAI_API_KEY="COPY/PASTE KEY"

AZURE_OPENAI_CHAT_DEPLOYMENT="gpt-4o-mini"
AZURE_OPENAI_EMBED_DEPLOYMENT="text-embedding-3-large"

CHUNK_SIZE=800
CHUNK_OVERLAP=120
MAX_CHARS_PER_FILE=80000
MAX_TOTAL_CHUNKS=2000
```

Figure 7

Next, you are going to fill the data folder with as many files as you want on a certain subject or something you want your chatbot to be able to answer. Then as indicated in the README you run the ingest.py and then the streamlit app (read README.md for detailed infos).

If you want to understand the code better I advise you to explore the app/rag.py where we will first search in data the right parts to build our context with embed_texts(), load_index(), search() and build_context(). The index lets us avoid repeatedly scanning the same parts of the text by providing a fast reference of the text and have a sort of reference that we can access faster.

After that, in the prompts.py, that you can modify, you have the prompt that will be sent to the LLM. With context, task, and format:

```
SYSTEM_RAG = '''
Tu es un assistant interne d'entreprise.
Réponds avec des fait et de manière sourcée.

Tu dois uniquement utiliser le contexte fourni et rien d'autre.
Sinon tu ne sais pas et c'est ok.
Fini avec une partie "Sources " qui donne les documents et les titres
'''


USER_RAG_TEMPLATE = """
Question: {question}
Contexte : {context}
Réponds en français, avec au moins 5 lignes et maximum 10 lignes. Ajoute "Sources:" avec les titres.
"""
```

Figure 8

The format is here SYSTEM_RAG and is composed of the persona (Company assistant) and the format (answer to the question, with detailed sources, in English and only using the given context)

Finally, we send the whole ‘context + question’ and then build the answer in rag.py with this function:

```
# make answer with index and OpenAI answer
def answer(question: str) -> Dict:
    snippets = search(question, TOP_K)
    context = build_context(snippets)

    user_prompt = USER_RAG_TEMPLATE.format(question=question, context=context)

    chat = client.chat.completions.create(
        model=CHAT_MODEL,
        messages=[
            {"role": "system", "content": SYSTEM_RAG},
            {"role": "user", "content": user_prompt},
        ],
        temperature=0.2,
    )

    content = chat.choices[0].message.content

    return {
        "answer": content,
        "sources": [
            {"title": s["title"], "source": s["source"], "score": s["score"]}
            for s in snippets
        ],
    }
```

Figure 9

Streamlit app

The Streamlit app is really basic, feel free to adapt it for your needs. So you can find the main file in ui/streamlit_app.py. The display is very simple and is built with this:

```
q = st.text_input( 'Quelle est ta question sur Ateme ou le stage')
if  st.button("Ask" ) or q:
    with st.spinner(' Processing...'):
        res = answer(q )

    st.markdown( res["answer" ] )


    with st.expander("Sources" ):
        for  s in res["sources" ]:
            st.write( f"- {s['title'  ]}")
```

Figure 10

The streamlit app running is explained in the README.md but if you already done all the imports and run the ingest.py, just run the following command: *streamlit run ui/streamlit_app.py* and a page will automatically open. If not just copy and paste the url in your terminal.

You should have this:

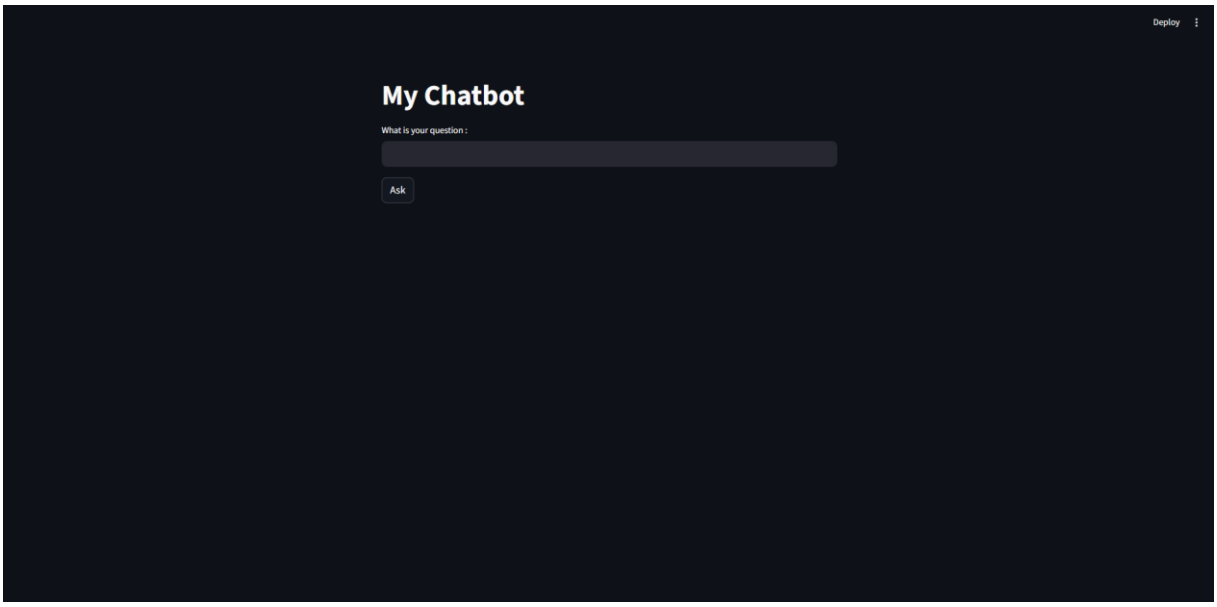

Figure 11

Now, you are ready to go with your own operational chatbot. You can add as much data as you want in the data folder (only certain document types such as .pdf, .txt, .md and .html). But after each addition in the data folder, make sure to re-run *python ingest.py* and then restart your Streamlit app; otherwise, the changes won't be taken into account.

Conclusion

In this project, we built a lightweight yet effective RAG-based chatbot powered by Azure OpenAI. The pipeline ingests and cleans your documents, chunks them, and indexes their embeddings for fast retrieval. At query time, the system retrieves the most relevant passages, composes a focused context, and asks the model to generate a grounded answer with cited sources. The result is an easily reproducible stack (Python, FAISS, FastAPI/Streamlit, Azure OpenAI) that you can adapt to new domains simply by updating the data folder and rebuilding the index. This approach strikes a practical balance between performance, control, and simplicity; ideal for internal knowledge assistants and prototypes that need reliable, source-backed responses.

Glossary

Chatbot: A program that simulates conversation with users and provides answers or assistance.

RAG (Retrieval-Augmented Generation): A method that retrieves relevant text snippets from your data and feeds them to a language model to produce grounded answers.

Embedding: A numerical vector representation of text that captures semantic meaning; used for similarity search.

FAISS: A high-performance library for efficient similarity search and clustering of dense vectors.

Index: A data structure (here, FAISS) that stores embeddings to enable fast nearest-neighbor search.

Chunking: Splitting long documents into smaller, overlapping pieces to improve retrieval quality.

Azure OpenAI: Microsoft’s managed service providing access to OpenAI models via Azure endpoints.

LLM (Large Language Model): A neural network trained on large corpora to understand and generate human-like text.

Token: The smallest text unit processed by an LLM (word, subword, or symbol).

Streamlit: A Python framework for building simple web apps and dashboards.

FastAPI: A modern Python web framework for building APIs quickly, used here for the backend endpoint.