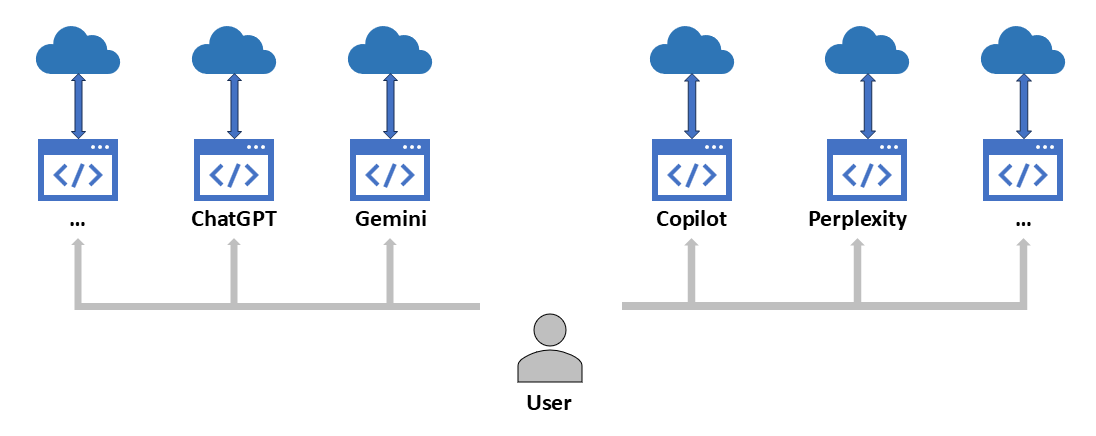


Secure, free, and personalized Generative AI RAG assistant on a personal computer in 3 steps

Link: <https://www.linkedin.com/pulse/secure-free-personalized-generative-ai-rag-assistant-personal-yogesh-v9xre>

Traditional public domain generative AI solutions like ChatGPT, Gemini, Copilot, Perplexity, and others are based on the information available on free, public domain sites on the internet.

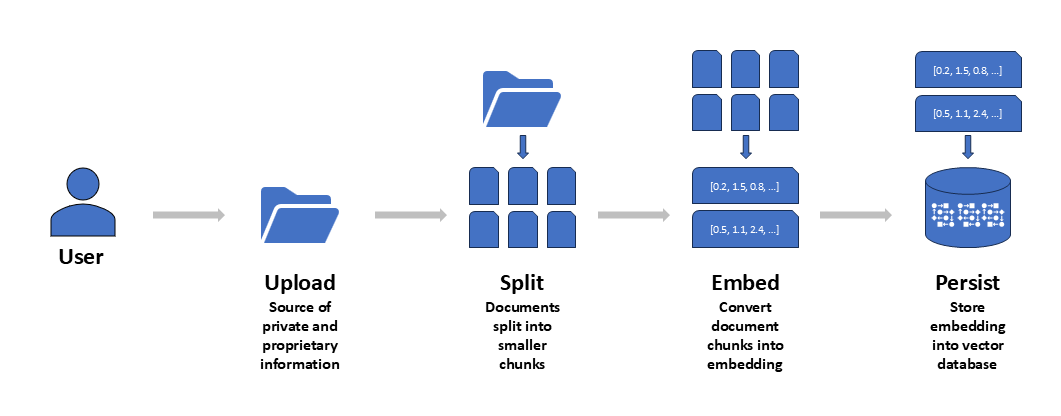


*Current state of traditional public domain generative AI solutions*

All the above solutions have a common limitation of not having the access proprietary information stored in private and secure documents used by organizations and individuals. This limitation stops them from answering the questions based on the information stored in private, secure, and proprietary documents.

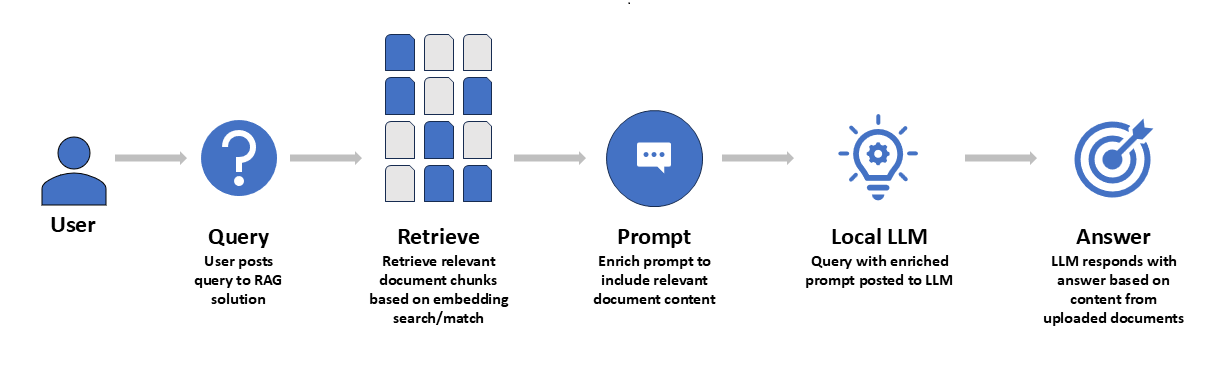
**Retrieval-augmented generation (RAG)** is an in-context learning technique that retrieves relevant information from a large dataset to augment the input context before generating a response or output. This approach aims to improve the accuracy of the generated content by leveraging external sources.

RAG solution has two steps. The first ***Indexing*** step loads, splits, embeds, and stores local documents into a local vector database.



*Indexing*

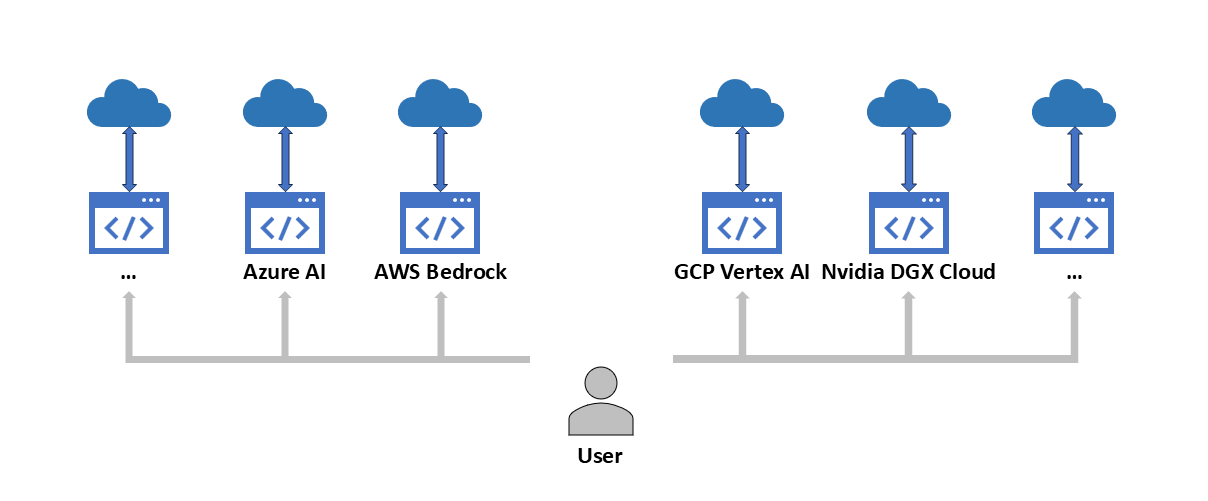
The second ***Retrieval and Generation*** step retrieves relevant documents from the local vector database and passes the content in prompt to the local LLM to generate the final answer.



*Retrieval and Generation*

**Current state of public domain RAG solutions**

Since the release of ChapGPT in 2022, we have been surrounded by an explosion of generative AI (GenAI) Retrieval-Augmented Generation (RAG) solutions available in the public domain. Most people use sites like Azure AI, AWS Bedrock, Google Cloud Vertex AI, Nvidia DGX Cloud, and others to create content using GenAI RAG solutions.



*Current situation of user consumption of public domain GenAI RAG solutions*

There are three key challenges with using public domain GenAI sites:

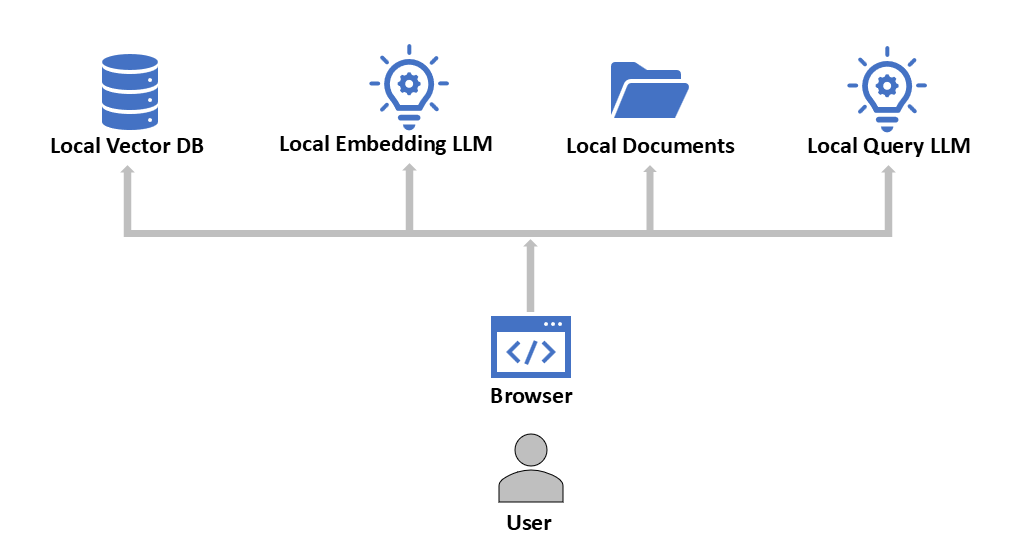
* **Security**: Users are at the mercy of the GenAI sites for the security, storage, and usage of the queries and other content posted on these sites. This is critical especially while sharing proprietary knowledge with the public domain sites.
* **Cost**: Most popular GenAI sites allow free content generation up to a certain limit and require a paid subscription for everything beyond the free limit.
* **Lack of Personalization**: The content generated by the GenAI sites is based on publicly available content on the Internet and is not personalized for the user.

While most people are aware of these challenges, the risks of exposing personal information and subsequent exploitation are high, especially when there are limited laws and regulations to protect consumers.

So, I decided to build a personalized GenAI RAG solution to run on a personal computer without worrying about data security, and expensive fees.

**Solution Overview**

The solution is simple. It is a web application running on a local machine that uses free and open-source LLMs and a free and open-source vector database running on local machine to provide answers to user queries based on local private and proprietary documents.



Secure, free, and personalized Generative AI RAG assistant on local computer

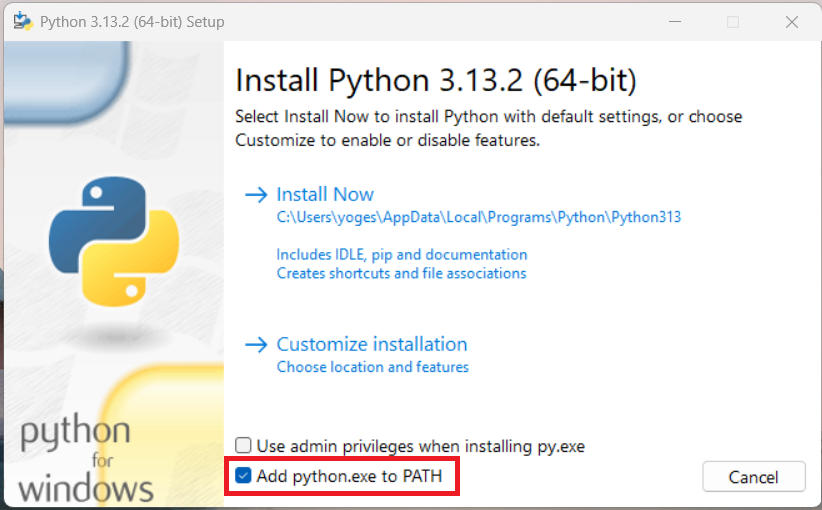
**Steps to create GenAI RAG assistant**

This section describes three steps to complete the setup of the above solution.

* Step #1: Install [Python](https://www.python.org/) (a free and open source high-level, general-purpose programming language) and install [ChromaDB](https://docs.trychroma.com/docs/overview/introduction) (a free and open source AI application vector database) and other necessary dependencies
* Step #2: Install [Ollama](https://ollama.com/) (a free and open source platform to run LLMs on a local machine) and download [Llama](https://www.llama.com/) (a free and open source LLM by Meta AI) and [nomic-embed-text](https://www.nomic.ai/blog/posts/nomic-embed-text-v1) (a high-performing open embedding model with a large token context window) on a local machine
* Step #3: Create a local web application using Python to access a local LLM instance

**Step #1: Install Python**

* Download and run the Python Windows installer from the [URL](https://www.python.org/downloads/windows)
* Follow the installer instructions to complete installation, and select the option to Add Python to PATH when asked during installation.



Python Installation Option Selection

**Step #2: Install Ollama and Llama LLM**

* Download and run the Ollama Windows installer from the [URL](https://ollama.com/download/windows)
* Download and install Llama and nomic-embed-text LLM by using the following command on Windows Command Prompt

<Code>

ollama pull nomic-embed-text

ollama pull llama3

</Code>

**Step #3: Create Web Application**

* Create a folder on a local machine and open the folder in Windows Command Prompt.
* Install Python libraries (streamlit, pandas, langchain, langchain-community, langchain-chroma, langchain-ollama, PyPDF2, python-docx, and openpyxl) using the following commands.

<Code>

python.exe -m pip install --upgrade pip

pip install streamlit pandas langchain langchain-community langchain-chroma langchain-ollama PyPDF2 python-docx openpyxl

</Code>

* Create a Python script named rag\_assistant.py and populate it with the following code for the web application.

<Code>

import streamlit as st

import os

import pandas as pd

import shutil

import time

import re

from langchain\_community.document\_loaders import TextLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain\_chroma import Chroma

from langchain\_ollama import OllamaEmbeddings

from langchain\_ollama import OllamaLLM

from langchain.chains import RetrievalQA

from langchain.prompts import PromptTemplate

from PyPDF2 import PdfReader

from docx import Document

from langchain\_core.documents import Document as LangChainDocument

# Initialize session state

if "vector\_store" not in st.session\_state:

st.session\_state.vector\_store = None

if "qa\_history" not in st.session\_state:

st.session\_state.qa\_history = []

if "processed\_files" not in st.session\_state:

st.session\_state.processed\_files = set()

# Persistent directory for Chroma

CHROMA\_PERSIST\_DIR = "./chroma\_db"

COLLECTION\_NAME = "rag\_collection"

# Embedding model

embeddings = OllamaEmbeddings(model="nomic-embed-text")

# Initialize or load vector store

def initialize\_vector\_store(force\_reset=False):

if force\_reset and os.path.exists(CHROMA\_PERSIST\_DIR):

shutil.rmtree(CHROMA\_PERSIST\_DIR, ignore\_errors=True)

if os.path.exists(CHROMA\_PERSIST\_DIR):

try:

vector\_store = Chroma(

persist\_directory=CHROMA\_PERSIST\_DIR,

embedding\_function=embeddings,

collection\_name=COLLECTION\_NAME

)

vector\_store.\_collection.count()

return vector\_store

except Exception as e:

st.warning(f"Failed to load vector store: {str(e)}. Resetting directory.")

shutil.rmtree(CHROMA\_PERSIST\_DIR, ignore\_errors=True)

return None

# Streamlit UI

st.title("Local free, secure and personalized Generative AI RAG assistant")

# File uploader

uploaded\_files = st.file\_uploader("Upload files", type=["txt", "pdf", "docx", "xlsx"], accept\_multiple\_files=True)

def preprocess\_text(text):

"""Normalize text for better embeddings and matching."""

text = re.sub(r'\s+', ' ', text) # Collapse multiple spaces/newlines into single space

text = text.strip() # Remove leading/trailing whitespace

return text.lower() # Convert to lowercase for consistent matching

def process\_file(file, file\_type):

"""Process a file and return LangChain documents with correct metadata."""

temp\_file = f"temp\_file\_{file.name}"

with open(temp\_file, "wb") as f:

f.write(file.getbuffer())

original\_filename = file.name

if file\_type == "txt":

loader = TextLoader(temp\_file)

documents = loader.load()

for doc in documents:

doc.page\_content = preprocess\_text(doc.page\_content)

doc.metadata = {"source": original\_filename}

elif file\_type == "pdf":

reader = PdfReader(temp\_file)

text = "".join(page.extract\_text() + " " for page in reader.pages if page.extract\_text())

text = preprocess\_text(text)

documents = [LangChainDocument(page\_content=text, metadata={"source": original\_filename})]

elif file\_type == "docx":

doc = Document(temp\_file)

text = " ".join(para.text for para in doc.paragraphs if para.text.strip()) # Join paragraphs with space

text = preprocess\_text(text)

documents = [LangChainDocument(page\_content=text, metadata={"source": original\_filename})]

elif file\_type == "xlsx":

df = pd.read\_excel(temp\_file)

text = " ".join(df.astype(str).agg(' '.join, axis=1)) # Convert to string and join rows with space

text = preprocess\_text(text)

documents = [LangChainDocument(page\_content=text, metadata={"source": original\_filename})]

os.remove(temp\_file)

return documents

# Process uploaded files

if uploaded\_files:

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=100)

for uploaded\_file in uploaded\_files:

if uploaded\_file.name not in st.session\_state.processed\_files:

try:

file\_type = uploaded\_file.name.split(".")[-1].lower()

documents = process\_file(uploaded\_file, file\_type)

chunks = text\_splitter.split\_documents(documents)

for chunk in chunks:

chunk.metadata = {"source": uploaded\_file.name}

if st.session\_state.vector\_store is None:

st.session\_state.vector\_store = Chroma.from\_documents(

documents=chunks,

embedding=embeddings,

collection\_name=COLLECTION\_NAME,

persist\_directory=CHROMA\_PERSIST\_DIR

)

st.success(f"Created new vector store with '{uploaded\_file.name}'")

else:

st.session\_state.vector\_store.add\_documents(chunks)

st.success(f"Added '{uploaded\_file.name}' to vector store")

st.session\_state.processed\_files.add(uploaded\_file.name)

except Exception as e:

st.error(f"Error processing '{uploaded\_file.name}': {str(e)}")

# Display processed files

if st.session\_state.processed\_files:

st.subheader("Processed Files")

for file\_name in st.session\_state.processed\_files:

st.write(f"- {file\_name}")

# Question input

st.subheader("Ask a Question")

with st.form(key="question\_form", clear\_on\_submit=True):

question = st.text\_input("Ask a question about the documents:")

submit\_button = st.form\_submit\_button(label="Submit")

if submit\_button and question:

if st.session\_state.vector\_store is None:

st.session\_state.vector\_store = initialize\_vector\_store()

if st.session\_state.vector\_store:

llm = OllamaLLM(model="llama3")

retriever = st.session\_state.vector\_store.as\_retriever(

search\_type="similarity",

search\_kwargs={"k": 5}

)

# Debug retrieved documents

with st.expander("Debug Information"):

count = st.session\_state.vector\_store.\_collection.count()

st.write(f"Total chunks in vector store: {count}")

retrieved\_docs = retriever.invoke(question)

st.write("Retrieved Documents:")

for i, doc in enumerate(retrieved\_docs):

source = doc.metadata.get("source", "Unknown")

st.write(f"Doc {i+1} from '{source}': {doc.page\_content[:200]}...")

# Prompt for stuffing all documents

prompt = PromptTemplate(

input\_variables=["context", "question"],

template=(

"You are a precise assistant answering based solely on the provided document content. "

"Extract the answer directly from the context below. If the context doesn’t contain the answer, respond only with 'Unable to answer based on documents.'\n\n"

"Context: {context}\n\n"

"Question: {question}\n\n"

"Answer:"

)

)

# QA chain with stuff

try:

qa\_chain = RetrievalQA.from\_chain\_type(

llm=llm,

chain\_type="stuff",

retriever=retriever,

return\_source\_documents=True,

chain\_type\_kwargs={

"prompt": prompt,

"document\_variable\_name": "context"

}

)

with st.spinner("Generating answer..."):

result = qa\_chain.invoke({"query": question})

answer = result["result"].strip()

source\_docs = result.get("source\_documents", [])

# Determine contributing sources by matching the exact answer or key phrase

contributing\_sources = set()

normalized\_answer = answer.lower().strip()

unable\_to\_answer = "unable to answer based on documents" in normalized\_answer

core\_answer = None # Initialize core\_answer outside the if block

if not unable\_to\_answer:

# Extract the core answer by removing preamble and quotes

core\_answer\_match = re.search(r'["“](.\*?)["”]$', normalized\_answer)

core\_answer = core\_answer\_match.group(1) if core\_answer\_match else normalized\_answer

core\_answer = re.sub(r'^according to.\*?:\s\*', '', core\_answer).strip()

# Debug the extracted core answer and document contents

with st.expander("Source Matching Debug"):

st.write(f"Core Answer to Match: '{core\_answer}'")

for i, doc in enumerate(source\_docs):

st.write(f"Doc {i+1} Content (first 200 chars): '{doc.page\_content[:200]}...'")

st.write(f"Doc {i+1} Source: '{doc.metadata.get('source', 'Unknown')}'")

# Look for the core answer in each document

for doc in source\_docs:

doc\_content = doc.page\_content.lower() # Already normalized in preprocess\_text

if core\_answer in doc\_content:

contributing\_sources.add(doc.metadata.get("source", "Unknown"))

# If no exact match, use significant keywords as fallback

if not contributing\_sources:

answer\_keywords = set(re.split(r'\W+', core\_answer)) - set(["the", "is", "of", "in", "a", "an", "according", "to"])

significant\_keywords = {kw for kw in answer\_keywords if len(kw) > 3}

for doc in source\_docs:

doc\_content = doc.page\_content.lower()

if any(keyword in doc\_content for keyword in significant\_keywords):

contributing\_sources.add(doc.metadata.get("source", "Unknown"))

# Debug LLM input and output

with st.expander("LLM Debug"):

context = "\n".join(doc.page\_content for doc in source\_docs)

st.write(f"Context passed to LLM:\n{context[:500]}..." if len(context) > 500 else context)

st.write(f"Final Answer: '{answer}'")

st.write(f"Core Answer Extracted: '{core\_answer if core\_answer else 'N/A'}'")

st.write(f"Contributing Sources: {contributing\_sources}")

history\_entry = {

"question": question,

"answer": answer,

"sources": list(contributing\_sources) if not unable\_to\_answer else []

}

st.session\_state.qa\_history.append(history\_entry)

except Exception as e:

st.error(f"Error initializing QA chain or generating answer: {str(e)}")

else:

st.error("No documents loaded. Please upload files first.")

# Display history

st.subheader("Question and Answer History")

for i, qa in enumerate(st.session\_state.qa\_history):

st.write(f"\*\*Q{i+1}:\*\* {qa['question']}")

st.write(f"\*\*A{i+1}:\*\* {qa['answer']}")

if qa["sources"]:

st.write(f"\*\*Sources:\*\* {', '.join(qa['sources'])}")

else:

st.write("\*\*Sources:\*\* None")

st.write("---")

# Reset button

if st.button("Reset Knowledge Base"):

try:

st.session\_state.vector\_store = initialize\_vector\_store(force\_reset=True)

st.session\_state.processed\_files = set()

st.session\_state.qa\_history = []

time.sleep(1)

st.success("Knowledge base reset successfully!")

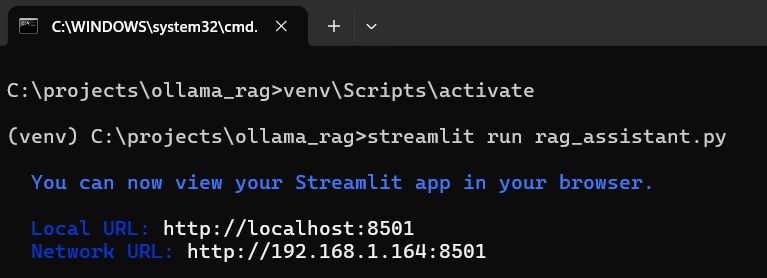
st.rerun()

except Exception as e:

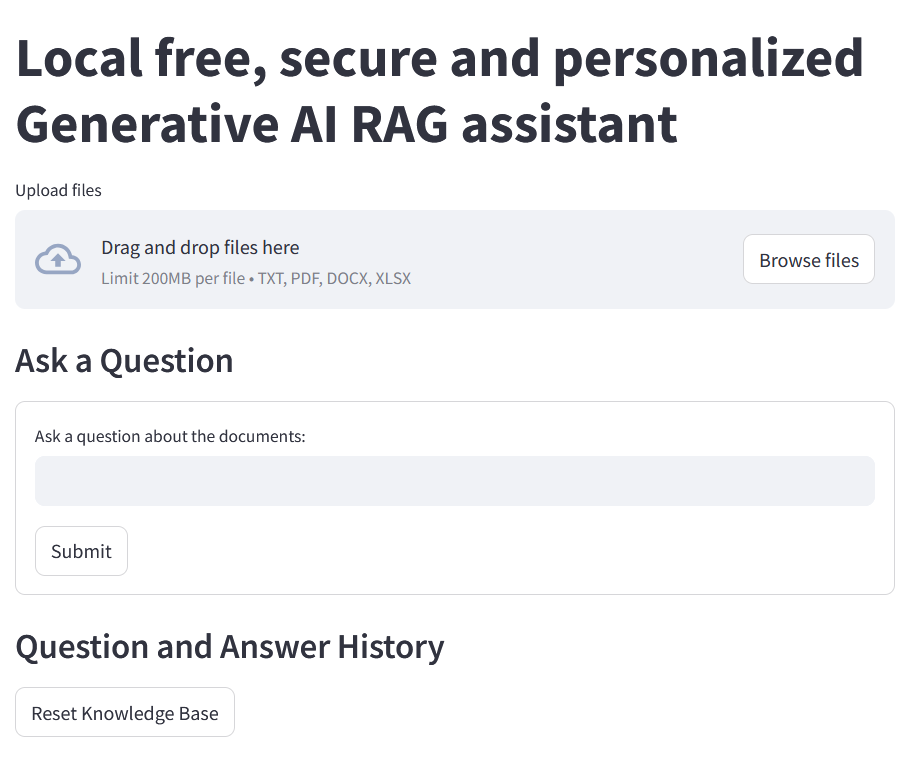
st.error(f"Error resetting: {str(e)}")

</Code>

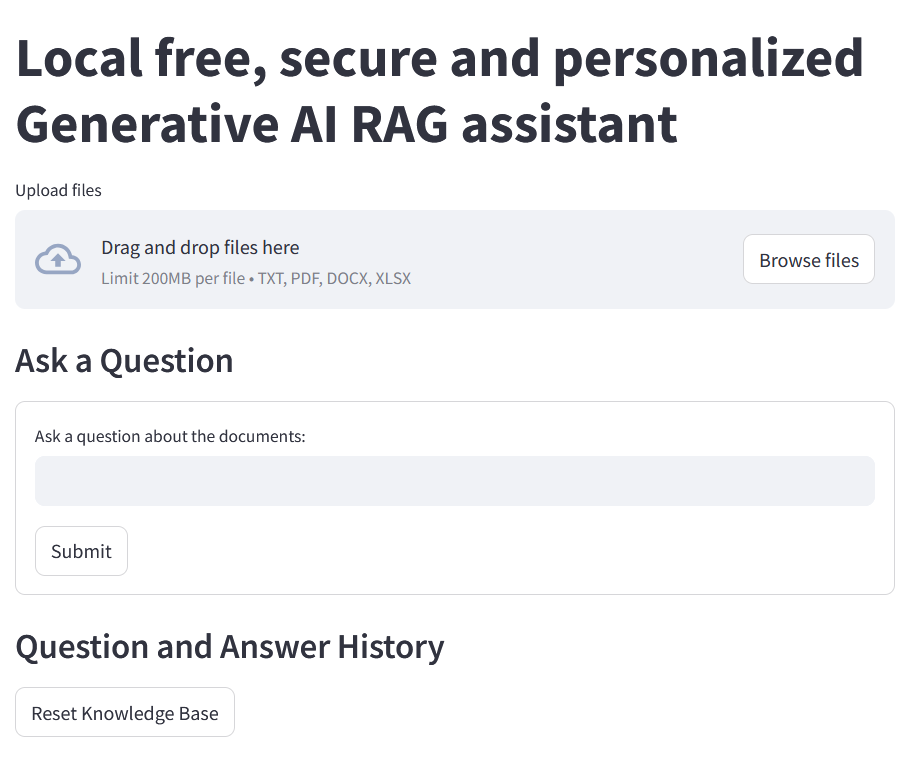
* Run local web application by running the following command on Windows Command Prompt.



* Access the application using the URL <http://localhost:8501>



* Test the application by uploading documents (Supported file types are TXT, PDF, DOCX, XLSX)



Local free, secure, and personalized generative AI Retrieval-augmented generation (RAG) assistant

I hope this guide helps you to build your local free, secure, and personalized generative AI Retrieval-augmented generation (RAG) assistant.

#python, #ollama, #llama, #nomic-embed-text, #chromadb, #generativeai, #genai, #rag

Git Repo: <https://github.com/yogeshsarkhot/gen-ai-rag-assistant>