give me the code in python for a basic rag implementation chatbot that can read pdf,docx,doc,ppt and give accurate output. also use llama and groq

I'll create a comprehensive RAG (Retrieval-Augmented Generation) chatbot implementation that can handle multiple document formats and uses Groq with Llama models. This will be a production-ready implementation with proper document processing and vector search capabilities.

RAG Chatbot with Multi-Format Document Support

Code

I've created a comprehensive RAG chatbot implementation that supports PDF, DOCX, DOC, and PPT files with Groq and Llama integration. Here are the key features:

**Key Components:**

1. **Document Processing**: Handles multiple formats:
   * PDF (using PyPDF2)
   * DOCX (using python-docx)
   * DOC (using docx2txt or win32com)
   * PPT/PPTX (using python-pptx)
2. **Text Chunking**: Intelligently splits documents into overlapping chunks for better retrieval
3. **Vector Storage**: Uses ChromaDB with sentence-transformers for embedding and similarity search
4. **Groq Integration**: Uses Groq API with Llama models for response generation

**Required Dependencies:**

bash

pip install groq chromadb sentence-transformers PyPDF2 python-docx python-pptx nltk tiktoken numpy docx2txt

For Windows users (DOC file support):

bash

pip install pywin32

**Setup Instructions:**

1. **Get a Groq API Key**: Sign up at [console.groq.com](https://console.groq.com/) and get your API key
2. **Set Environment Variable**:

bash

export GROQ\_API\_KEY="your\_groq\_api\_key\_here"

1. **Run the chatbot**:

python

python rag\_chatbot.py

**Usage Examples:**

python

*# Initialize chatbot*

chatbot = RAGChatbot("your\_groq\_api\_key")

*# Add documents*

chatbot.add\_documents(["document1.pdf", "presentation.pptx", "report.docx"])

*# Chat with the bot*

response = await chatbot.chat("What are the main findings in the report?")

print(response)

**Features:**

* **Multi-format Support**: PDF, DOCX, DOC, PPT, PPTX
* **Intelligent Chunking**: Sentence-aware text splitting with overlap
* **Vector Search**: Semantic similarity search using embeddings
* **Context-Aware Responses**: Cites sources and page numbers
* **Conversation History**: Maintains context across queries
* **Error Handling**: Robust error handling for file processing
* **Persistent Storage**: ChromaDB persists your document embeddings

**Advanced Configuration:**

You can customize various parameters:

* chunk\_size: Size of text chunks (default: 1000 tokens)
* overlap: Overlap between chunks (default: 100 tokens)
* model\_name: Groq model to use (default: "llama3-8b-8192")
* k: Number of documents to retrieve (default: 5)

The chatbot provides accurate responses by retrieving relevant document chunks and using them as context for the Llama model through Groq's API.

import os

import logging

from typing import List, Dict, Any, Optional

from dataclasses import dataclass

import asyncio

from pathlib import Path

*# Document processing*

import PyPDF2

import docx

import win32com.client as win32 *# For .doc files*

from pptx import Presentation

import mammoth *# Alternative for .docx*

*# Vector embeddings and similarity search*

import chromadb

from sentence\_transformers import SentenceTransformer

import numpy as np

*# Groq API*

from groq import Groq

import tiktoken

*# Text processing*

import re

from nltk.tokenize import sent\_tokenize, word\_tokenize

import nltk

*# Download required NLTK data*

try:

nltk.data.find('tokenizers/punkt')

except LookupError:

nltk.download('punkt')

*# Configure logging*

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

@dataclass

class Document:

"""Represents a processed document with metadata."""

content: str

source: str

page\_number: Optional[int] = None

chunk\_id: Optional[str] = None

metadata: Dict[str, Any] = None

class DocumentProcessor:

"""Handles processing of different document formats."""

def \_\_init\_\_(self):

self.supported\_formats = ['.pdf', '.docx', '.doc', '.ppt', '.pptx']

def process\_document(self, file\_path: str) -> List[Document]:

"""Process a document and return a list of Document objects."""

file\_path = Path(file\_path)

if not file\_path.exists():

raise FileNotFoundError(f"File not found: {file\_path}")

extension = file\_path.suffix.lower()

if extension not in self.supported\_formats:

raise ValueError(f"Unsupported file format: {extension}")

logger.info(f"Processing {extension} file: {file\_path}")

if extension == '.pdf':

return self.\_process\_pdf(file\_path)

elif extension == '.docx':

return self.\_process\_docx(file\_path)

elif extension == '.doc':

return self.\_process\_doc(file\_path)

elif extension in ['.ppt', '.pptx']:

return self.\_process\_ppt(file\_path)

def \_process\_pdf(self, file\_path: Path) -> List[Document]:

"""Process PDF files."""

documents = []

try:

with open(file\_path, 'rb') as file:

pdf\_reader = PyPDF2.PdfReader(file)

for page\_num, page in enumerate(pdf\_reader.pages, 1):

text = page.extract\_text()

if text.strip(): *# Only add non-empty pages*

documents.append(Document(

content=text,

source=str(file\_path),

page\_number=page\_num,

metadata={'file\_type': 'pdf', 'total\_pages': len(pdf\_reader.pages)}

))

except Exception as e:

logger.error(f"Error processing PDF {file\_path}: {e}")

raise

return documents

def \_process\_docx(self, file\_path: Path) -> List[Document]:

"""Process DOCX files."""

try:

doc = docx.Document(file\_path)

content = []

for paragraph in doc.paragraphs:

if paragraph.text.strip():

content.append(paragraph.text)

*# Also extract text from tables*

for table in doc.tables:

for row in table.rows:

row\_text = []

for cell in row.cells:

if cell.text.strip():

row\_text.append(cell.text.strip())

if row\_text:

content.append(" | ".join(row\_text))

full\_text = "\n".join(content)

return [Document(

content=full\_text,

source=str(file\_path),

metadata={'file\_type': 'docx'}

)]

except Exception as e:

logger.error(f"Error processing DOCX {file\_path}: {e}")

raise

def \_process\_doc(self, file\_path: Path) -> List[Document]:

"""Process DOC files using win32com (Windows only)."""

try:

*# Alternative approach using python-docx2txt for cross-platform compatibility*

try:

import docx2txt

text = docx2txt.process(str(file\_path))

return [Document(

content=text,

source=str(file\_path),

metadata={'file\_type': 'doc'}

)]

except ImportError:

*# Fallback to win32com (Windows only)*

word = win32.Dispatch("Word.Application")

word.Visible = False

doc = word.Documents.Open(str(file\_path))

text = doc.Content.Text

doc.Close()

word.Quit()

return [Document(

content=text,

source=str(file\_path),

metadata={'file\_type': 'doc'}

)]

except Exception as e:

logger.error(f"Error processing DOC {file\_path}: {e}")

raise

def \_process\_ppt(self, file\_path: Path) -> List[Document]:

"""Process PPT/PPTX files."""

try:

prs = Presentation(file\_path)

documents = []

for slide\_num, slide in enumerate(prs.slides, 1):

slide\_text = []

for shape in slide.shapes:

if hasattr(shape, "text") and shape.text.strip():

slide\_text.append(shape.text)

if slide\_text:

content = "\n".join(slide\_text)

documents.append(Document(

content=content,

source=str(file\_path),

page\_number=slide\_num,

metadata={'file\_type': 'pptx', 'slide\_number': slide\_num}

))

return documents

except Exception as e:

logger.error(f"Error processing PPT {file\_path}: {e}")

raise

class TextChunker:

"""Handles text chunking for better retrieval."""

def \_\_init\_\_(self, chunk\_size: int = 1000, overlap: int = 100):

self.chunk\_size = chunk\_size

self.overlap = overlap

self.tokenizer = tiktoken.get\_encoding("cl100k\_base")

def chunk\_documents(self, documents: List[Document]) -> List[Document]:

"""Chunk documents into smaller pieces."""

chunked\_docs = []

for doc in documents:

chunks = self.\_chunk\_text(doc.content)

for i, chunk in enumerate(chunks):

chunked\_doc = Document(

content=chunk,

source=doc.source,

page\_number=doc.page\_number,

chunk\_id=f"{doc.source}\_chunk\_{i}",

metadata={\*\*(doc.metadata or {}), 'chunk\_index': i, 'total\_chunks': len(chunks)}

)

chunked\_docs.append(chunked\_doc)

return chunked\_docs

def \_chunk\_text(self, text: str) -> List[str]:

"""Chunk text into overlapping segments."""

*# First try sentence-based chunking*

sentences = sent\_tokenize(text)

chunks = []

current\_chunk = []

current\_length = 0

for sentence in sentences:

sentence\_length = len(self.tokenizer.encode(sentence))

if current\_length + sentence\_length > self.chunk\_size and current\_chunk:

chunks.append(" ".join(current\_chunk))

*# Calculate overlap*

overlap\_sentences = []

overlap\_length = 0

for sent in reversed(current\_chunk):

sent\_length = len(self.tokenizer.encode(sent))

if overlap\_length + sent\_length <= self.overlap:

overlap\_sentences.insert(0, sent)

overlap\_length += sent\_length

else:

break

current\_chunk = overlap\_sentences

current\_length = overlap\_length

current\_chunk.append(sentence)

current\_length += sentence\_length

if current\_chunk:

chunks.append(" ".join(current\_chunk))

return chunks

class VectorStore:

"""Handles vector storage and retrieval using ChromaDB."""

def \_\_init\_\_(self, collection\_name: str = "rag\_documents", persist\_directory: str = "./chroma\_db"):

self.client = chromadb.PersistentClient(path=persist\_directory)

self.collection\_name = collection\_name

self.embedding\_model = SentenceTransformer('all-MiniLM-L6-v2')

*# Create or get collection*

try:

self.collection = self.client.get\_collection(name=collection\_name)

logger.info(f"Loaded existing collection: {collection\_name}")

except ValueError:

self.collection = self.client.create\_collection(

name=collection\_name,

metadata={"hnsw:space": "cosine"}

)

logger.info(f"Created new collection: {collection\_name}")

def add\_documents(self, documents: List[Document]):

"""Add documents to the vector store."""

if not documents:

return

*# Prepare data for ChromaDB*

ids = []

embeddings = []

metadatas = []

contents = []

for doc in documents:

doc\_id = doc.chunk\_id or f"{doc.source}\_{hash(doc.content)}"

ids.append(doc\_id)

contents.append(doc.content)

*# Generate embeddings*

embedding = self.embedding\_model.encode(doc.content)

embeddings.append(embedding.tolist())

*# Prepare metadata*

metadata = {

'source': doc.source,

'page\_number': doc.page\_number or 0,

\*\*(doc.metadata or {})

}

metadatas.append(metadata)

*# Add to collection*

self.collection.add(

ids=ids,

embeddings=embeddings,

metadatas=metadatas,

documents=contents

)

logger.info(f"Added {len(documents)} documents to vector store")

def similarity\_search(self, query: str, k: int = 5) -> List[Dict[str, Any]]:

"""Search for similar documents."""

query\_embedding = self.embedding\_model.encode(query)

results = self.collection.query(

query\_embeddings=[query\_embedding.tolist()],

n\_results=k

)

*# Format results*

formatted\_results = []

for i in range(len(results['ids'][0])):

formatted\_results.append({

'id': results['ids'][0][i],

'content': results['documents'][0][i],

'metadata': results['metadatas'][0][i],

'distance': results['distances'][0][i] if 'distances' in results else None

})

return formatted\_results

class RAGChatbot:

"""Main RAG chatbot class."""

def \_\_init\_\_(self, groq\_api\_key: str, model\_name: str = "llama3-8b-8192"):

self.groq\_client = Groq(api\_key=groq\_api\_key)

self.model\_name = model\_name

self.document\_processor = DocumentProcessor()

self.text\_chunker = TextChunker()

self.vector\_store = VectorStore()

*# Conversation history*

self.conversation\_history = []

def add\_documents(self, file\_paths: List[str]):

"""Add documents to the knowledge base."""

all\_documents = []

for file\_path in file\_paths:

try:

documents = self.document\_processor.process\_document(file\_path)

chunked\_documents = self.text\_chunker.chunk\_documents(documents)

all\_documents.extend(chunked\_documents)

logger.info(f"Processed {len(documents)} documents from {file\_path}")

except Exception as e:

logger.error(f"Failed to process {file\_path}: {e}")

continue

if all\_documents:

self.vector\_store.add\_documents(all\_documents)

logger.info(f"Added {len(all\_documents)} document chunks to knowledge base")

return len(all\_documents)

def \_create\_context\_prompt(self, query: str, retrieved\_docs: List[Dict[str, Any]]) -> str:

"""Create a context-aware prompt for the LLM."""

context\_parts = []

for i, doc in enumerate(retrieved\_docs, 1):

source\_info = f"Source: {doc['metadata'].get('source', 'Unknown')}"

if doc['metadata'].get('page\_number'):

source\_info += f", Page: {doc['metadata']['page\_number']}"

context\_parts.append(f"Document {i}:\n{source\_info}\nContent: {doc['content']}\n")

context = "\n".join(context\_parts)

prompt = f"""Based on the following documents, please answer the user's question accurately and comprehensively.

CONTEXT DOCUMENTS:

{context}

USER QUESTION: {query}

Instructions:

- Use only the information provided in the context documents

- If the answer is not found in the documents, clearly state that

- Cite the source document(s) when providing information

- Be concise but thorough in your response

- If multiple documents contain relevant information, synthesize them appropriately

ANSWER:"""

return prompt

async def chat(self, query: str, use\_history: bool = True) -> str:

"""Process a chat query using RAG."""

try:

*# Retrieve relevant documents*

retrieved\_docs = self.vector\_store.similarity\_search(query, k=5)

if not retrieved\_docs:

return "I don't have any relevant information in my knowledge base to answer your question. Please add some documents first."

*# Create context prompt*

prompt = self.\_create\_context\_prompt(query, retrieved\_docs)

*# Prepare messages for Groq*

messages = []

*# Add conversation history if requested*

if use\_history and self.conversation\_history:

messages.extend(self.conversation\_history[-10:]) *# Last 10 messages*

messages.append({"role": "user", "content": prompt})

*# Generate response using Groq*

completion = self.groq\_client.chat.completions.create(

model=self.model\_name,

messages=messages,

max\_tokens=1024,

temperature=0.1,

top\_p=0.9,

stream=False

)

response = completion.choices[0].message.content

*# Update conversation history*

self.conversation\_history.extend([

{"role": "user", "content": query},

{"role": "assistant", "content": response}

])

*# Keep history manageable*

if len(self.conversation\_history) > 20:

self.conversation\_history = self.conversation\_history[-20:]

return response

except Exception as e:

logger.error(f"Error in chat: {e}")

return f"Sorry, I encountered an error while processing your query: {str(e)}"

def clear\_history(self):

"""Clear conversation history."""

self.conversation\_history = []

logger.info("Conversation history cleared")

*# Example usage and CLI interface*

def main():

"""Example usage of the RAG chatbot."""

*# Initialize the chatbot*

GROQ\_API\_KEY = os.getenv("GROQ\_API\_KEY")

if not GROQ\_API\_KEY:

print("Please set your GROQ\_API\_KEY environment variable")

return

chatbot = RAGChatbot(GROQ\_API\_KEY)

print("RAG Chatbot initialized!")

print("Commands:")

print("- 'add <file\_path>' to add documents")

print("- 'clear' to clear conversation history")

print("- 'quit' to exit")

print("- Or just type your question\n")

while True:

user\_input = input("You: ").strip()

if user\_input.lower() == 'quit':

break

elif user\_input.lower() == 'clear':

chatbot.clear\_history()

print("Conversation history cleared.")

continue

elif user\_input.lower().startswith('add '):

file\_path = user\_input[4:].strip()

try:

count = chatbot.add\_documents([file\_path])

print(f"Added {count} document chunks to knowledge base.")

except Exception as e:

print(f"Error adding document: {e}")

continue

elif user\_input:

response = asyncio.run(chatbot.chat(user\_input))

print(f"Assistant: {response}\n")

if \_\_name\_\_ == "\_\_main\_\_":

main()

give me the code in python for a basic rag implementation chatbot that can read pdf,docx,doc,ppt and give accurate output. also use llama and groq

I'll create a comprehensive RAG (Retrieval-Augmented Generation) chatbot implementation that can handle multiple document formats and uses Groq with Llama models. This will be a production-ready implementation with proper document processing and vector search capabilities.

RAG Chatbot with Multi-Format Document Support

Code

I've created a comprehensive RAG chatbot implementation that supports PDF, DOCX, DOC, and PPT files with Groq and Llama integration. Here are the key features:

**Key Components:**

1. **Document Processing**: Handles multiple formats:
   * PDF (using PyPDF2)
   * DOCX (using python-docx)
   * DOC (using docx2txt or win32com)
   * PPT/PPTX (using python-pptx)
2. **Text Chunking**: Intelligently splits documents into overlapping chunks for better retrieval
3. **Vector Storage**: Uses ChromaDB with sentence-transformers for embedding and similarity search
4. **Groq Integration**: Uses Groq API with Llama models for response generation

**Required Dependencies:**

bash

pip install groq chromadb sentence-transformers PyPDF2 python-docx python-pptx nltk tiktoken numpy docx2txt

For Windows users (DOC file support):

bash

pip install pywin32

**Setup Instructions:**

1. **Get a Groq API Key**: Sign up at [console.groq.com](https://console.groq.com/) and get your API key
2. **Set Environment Variable**:

bash

export GROQ\_API\_KEY="your\_groq\_api\_key\_here"

1. **Run the chatbot**:

python

python rag\_chatbot.py

**Usage Examples:**

python

*# Initialize chatbot*

chatbot = RAGChatbot("your\_groq\_api\_key")

*# Add documents*

chatbot.add\_documents(["document1.pdf", "presentation.pptx", "report.docx"])

*# Chat with the bot*

response = await chatbot.chat("What are the main findings in the report?")

print(response)

**Features:**

* **Multi-format Support**: PDF, DOCX, DOC, PPT, PPTX
* **Intelligent Chunking**: Sentence-aware text splitting with overlap
* **Vector Search**: Semantic similarity search using embeddings
* **Context-Aware Responses**: Cites sources and page numbers
* **Conversation History**: Maintains context across queries
* **Error Handling**: Robust error handling for file processing
* **Persistent Storage**: ChromaDB persists your document embeddings

**Advanced Configuration:**

You can customize various parameters:

* chunk\_size: Size of text chunks (default: 1000 tokens)
* overlap: Overlap between chunks (default: 100 tokens)
* model\_name: Groq model to use (default: "llama3-8b-8192")
* k: Number of documents to retrieve (default: 5)

The chatbot provides accurate responses by retrieving relevant document chunks and using them as context for the Llama model through Groq's API.

how do i use it with vscode

I'll show you how to set up and use this RAG chatbot in VS Code with a complete step-by-step guide.

VS Code Setup Guide for RAG Chatbot

Document

I've created a comprehensive VS Code setup guide for your RAG chatbot. Here's a quick summary of how to get started:

**Quick Start Steps:**

1. **Create a new folder** for your project in VS Code
2. **Install Python extension** in VS Code
3. **Set up virtual environment**:

bash

python -m venv rag\_env

*# Windows: rag\_env\Scripts\activate*

*# Mac/Linux: source rag\_env/bin/activate*

1. **Create requirements.txt** and install dependencies:

bash

pip install -r requirements.txt

1. **Get your Groq API key** from [console.groq.com](https://console.groq.com/)
2. **Create .env file** with your API key:

GROQ\_API\_KEY=your\_api\_key\_here

1. **Copy the RAG chatbot code** into src/rag\_chatbot.py

**Three Ways to Run It:**

**1. Command Line (Simplest)**

* Press `Ctrl+`` to open terminal
* Run: python src/rag\_chatbot.py

**2. Streamlit Web Interface (Recommended)**

* Create the streamlit\_app.py file from the guide
* Run: streamlit run streamlit\_app.py
* Opens in your browser with a nice UI for uploading docs and chatting

**3. VS Code Debugger**

* Press F5 to debug
* Set breakpoints by clicking line numbers
* Great for development and troubleshooting

**Key VS Code Features You'll Love:**

* **IntelliSense**: Auto-completion as you type
* **Integrated Terminal**: No need to switch windows
* **Debug Mode**: Step through code line by line
* **Git Integration**: Version control built-in
* **File Upload**: Drag and drop documents into the project

The Streamlit web interface is probably the best way to use it - it gives you a clean UI where you can upload documents and chat with them directly in your browser!

Would you like me to help you with any specific part of the setup, or do you have questions about getting it running?

**VS Code Setup Guide for RAG Chatbot**

**Step 1: Install Required VS Code Extensions**

1. Open VS Code
2. Go to Extensions (Ctrl+Shift+X or Cmd+Shift+X)
3. Install these essential extensions:
   * **Python** (by Microsoft) - For Python development
   * **Pylance** (by Microsoft) - Python language server
   * **Python Debugger** (by Microsoft) - For debugging
   * **Jupyter** (by Microsoft) - If you want to use notebooks
   * **autoDocstring** (optional) - For generating docstrings

**Step 2: Create Project Structure**

Create a new folder for your RAG chatbot project:

rag-chatbot/

├── src/

│ ├── \_\_init\_\_.py

│ ├── rag\_chatbot.py # Main chatbot code

│ ├── document\_processor.py # Document processing logic

│ └── vector\_store.py # Vector storage logic

├── documents/ # Folder for your documents

├── data/

│ └── chroma\_db/ # ChromaDB will create this

├── tests/ # Unit tests

├── .env # Environment variables

├── .gitignore # Git ignore file

├── requirements.txt # Python dependencies

├── config.py # Configuration settings

└── README.md # Project documentation

**Step 3: Set Up Python Environment**

**Option A: Using VS Code's Built-in Terminal**

1. Open VS Code terminal (Ctrl+ or Cmd+)
2. Create a virtual environment:

bash

python -m venv rag\_env

1. Activate the virtual environment:
   * **Windows**: rag\_env\Scripts\activate
   * **Mac/Linux**: source rag\_env/bin/activate

**Option B: Using VS Code Python Interpreter**

1. Press Ctrl+Shift+P (or Cmd+Shift+P on Mac)
2. Type "Python: Create Environment"
3. Select "Venv"
4. Choose your Python interpreter
5. VS Code will create and activate the environment automatically

**Step 4: Install Dependencies**

Create a requirements.txt file:

txt

groq>=0.4.0

chromadb>=0.4.0

sentence-transformers>=2.2.0

PyPDF2>=3.0.0

python-docx>=0.8.11

python-pptx>=0.6.21

nltk>=3.8

tiktoken>=0.5.0

numpy>=1.24.0

docx2txt>=0.8

python-dotenv>=1.0.0

streamlit>=1.28.0

gradio>=3.50.0

Install dependencies:

bash

pip install -r requirements.txt

For Windows users (DOC file support):

bash

pip install pywin32

**Step 5: Environment Configuration**

Create a .env file in your project root:

env

GROQ\_API\_KEY=your\_groq\_api\_key\_here

MODEL\_NAME=llama3-8b-8192

CHUNK\_SIZE=1000

OVERLAP=100

VECTOR\_DB\_PATH=./data/chroma\_db

Create a config.py file:

python

import os

from dotenv import load\_dotenv

load\_dotenv()

*# API Configuration*

GROQ\_API\_KEY = os.getenv("GROQ\_API\_KEY")

MODEL\_NAME = os.getenv("MODEL\_NAME", "llama3-8b-8192")

*# Text Processing Configuration*

CHUNK\_SIZE = int(os.getenv("CHUNK\_SIZE", "1000"))

OVERLAP = int(os.getenv("OVERLAP", "100"))

*# Database Configuration*

VECTOR\_DB\_PATH = os.getenv("VECTOR\_DB\_PATH", "./data/chroma\_db")

*# Validation*

if not GROQ\_API\_KEY:

raise ValueError("GROQ\_API\_KEY environment variable is required")

**Step 6: VS Code Workspace Settings**

Create a .vscode/settings.json file for project-specific settings:

json

{

"python.defaultInterpreterPath": "./rag\_env/bin/python",

"python.terminal.activateEnvironment": true,

"python.formatting.provider": "black",

"python.linting.enabled": true,

"python.linting.pylintEnabled": true,

"python.testing.pytestEnabled": true,

"python.testing.unittestEnabled": false,

"files.associations": {

"\*.py": "python"

},

"python.analysis.autoImportCompletions": true,

"python.analysis.typeCheckingMode": "basic"

}

Create a .vscode/launch.json file for debugging:

json

{

"version": "0.2.0",

"configurations": [

{

"name": "Python: Current File",

"type": "python",

"request": "launch",

"program": "${file}",

"console": "integratedTerminal",

"justMyCode": true,

"envFile": "${workspaceFolder}/.env"

},

{

"name": "RAG Chatbot CLI",

"type": "python",

"request": "launch",

"program": "${workspaceFolder}/src/rag\_chatbot.py",

"console": "integratedTerminal",

"envFile": "${workspaceFolder}/.env"

},

{

"name": "Streamlit App",

"type": "python",

"request": "launch",

"module": "streamlit",

"args": ["run", "${workspaceFolder}/streamlit\_app.py"],

"console": "integratedTerminal",

"envFile": "${workspaceFolder}/.env"

}

]

}

**Step 7: Create a Streamlit Web Interface**

Create streamlit\_app.py for a web interface:

python

import streamlit as st

import asyncio

import os

from src.rag\_chatbot import RAGChatbot

from config import GROQ\_API\_KEY, MODEL\_NAME

st.set\_page\_config(

page\_title="RAG Chatbot",

page\_icon="🤖",

layout="wide"

)

@st.cache\_resource

def initialize\_chatbot():

return RAGChatbot(GROQ\_API\_KEY, MODEL\_NAME)

def main():

st.title("🤖 RAG Document Chatbot")

st.sidebar.title("Document Management")

*# Initialize chatbot*

chatbot = initialize\_chatbot()

*# Document upload section*

uploaded\_files = st.sidebar.file\_uploader(

"Upload documents",

type=['pdf', 'docx', 'doc', 'ppt', 'pptx'],

accept\_multiple\_files=True

)

if uploaded\_files:

if st.sidebar.button("Process Documents"):

with st.sidebar.spinner("Processing documents..."):

*# Save uploaded files temporarily*

temp\_paths = []

for uploaded\_file in uploaded\_files:

temp\_path = f"temp\_{uploaded\_file.name}"

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

f.write(uploaded\_file.getvalue())

temp\_paths.append(temp\_path)

*# Process documents*

count = chatbot.add\_documents(temp\_paths)

st.sidebar.success(f"Added {count} document chunks!")

*# Clean up temp files*

for temp\_path in temp\_paths:

os.remove(temp\_path)

*# Chat interface*

st.subheader("Chat with your documents")

*# Initialize chat history*

if "messages" not in st.session\_state:

st.session\_state.messages = []

*# Display chat history*

for message in st.session\_state.messages:

with st.chat\_message(message["role"]):

st.write(message["content"])

*# Chat input*

if prompt := st.chat\_input("Ask a question about your documents"):

*# Add user message to chat history*

st.session\_state.messages.append({"role": "user", "content": prompt})

with st.chat\_message("user"):

st.write(prompt)

*# Generate response*

with st.chat\_message("assistant"):

with st.spinner("Thinking..."):

response = asyncio.run(chatbot.chat(prompt))

st.write(response)

*# Add assistant response to chat history*

st.session\_state.messages.append({"role": "assistant", "content": response})

*# Sidebar controls*

if st.sidebar.button("Clear Chat History"):

st.session\_state.messages = []

chatbot.clear\_history()

st.rerun()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Step 8: Running the Chatbot in VS Code**

**Method 1: Command Line Interface**

1. Open VS Code terminal
2. Make sure your virtual environment is activated
3. Run:

bash

python src/rag\_chatbot.py

**Method 2: Using VS Code Debugger**

1. Open src/rag\_chatbot.py
2. Press F5 or go to Run → Start Debugging
3. Select "RAG Chatbot CLI" configuration

**Method 3: Streamlit Web Interface**

1. Run in terminal:

bash

streamlit run streamlit\_app.py

1. Or use the debugger with "Streamlit App" configuration

**Method 4: Jupyter Notebook (Optional)**

Create notebook\_example.ipynb:

python

*# Cell 1: Setup*

import asyncio

from src.rag\_chatbot import RAGChatbot

from config import GROQ\_API\_KEY

*# Initialize chatbot*

chatbot = RAGChatbot(GROQ\_API\_KEY)

*# Cell 2: Add documents*

document\_paths = [

"documents/your\_document.pdf",

"documents/your\_presentation.pptx"

]

count = chatbot.add\_documents(document\_paths)

print(f"Added {count} document chunks to knowledge base")

*# Cell 3: Chat*

query = "What are the main topics discussed in the documents?"

response = await chatbot.chat(query)

print(f"Response: {response}")

**Step 9: VS Code Tips and Tricks**

**Keyboard Shortcuts**

* Ctrl+Shift+P: Command Palette
* `Ctrl+``: Toggle Terminal
* F5: Start Debugging
* Ctrl+Shift+D: Debug View
* Ctrl+Shift+E: Explorer View
* Ctrl+K, Ctrl+S: Keyboard Shortcuts

**Useful VS Code Features**

1. **IntelliSense**: Auto-completion for Python code
2. **Debugging**: Set breakpoints by clicking line numbers
3. **Integrated Terminal**: Run commands without leaving VS Code
4. **Git Integration**: Built-in version control
5. **Extensions**: Python Test Explorer, GitLens, etc.

**Code Navigation**

* Ctrl+Click: Go to definition
* F12: Go to definition
* Shift+F12: Find all references
* Ctrl+Shift+O: Go to symbol in file
* Ctrl+T: Go to symbol in workspace

**Step 10: Testing and Debugging**

Create tests/test\_chatbot.py:

python

import unittest

import asyncio

from unittest.mock import Mock, patch

from src.rag\_chatbot import RAGChatbot

class TestRAGChatbot(unittest.TestCase):

def setUp(self):

self.chatbot = RAGChatbot("fake\_api\_key")

def test\_initialization(self):

self.assertIsNotNone(self.chatbot)

self.assertEqual(self.chatbot.model\_name, "llama3-8b-8192")

@patch('src.rag\_chatbot.DocumentProcessor')

def test\_add\_documents(self, mock\_processor):

mock\_processor.return\_value.process\_document.return\_value = []

count = self.chatbot.add\_documents(["test.pdf"])

self.assertEqual(count, 0)

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

Run tests in VS Code:

1. Install pytest: pip install pytest
2. Configure testing in VS Code settings
3. Use the Test Explorer to run tests

**Step 11: Deployment Options**

**Local Development**

* Use the CLI or Streamlit interface
* Great for testing and development

**Production Deployment**

* Deploy Streamlit app to Streamlit Cloud
* Use Docker for containerization
* Deploy to cloud platforms (AWS, GCP, Azure)

**Troubleshooting Common Issues**

1. **Import Errors**: Make sure your virtual environment is activated
2. **API Key Issues**: Check your .env file and ensure GROQ\_API\_KEY is set
3. **Package Installation**: Use pip list to verify installed packages
4. **Python Interpreter**: Use Ctrl+Shift+P → "Python: Select Interpreter"
5. **File Paths**: Use absolute paths or ensure files are in the correct directory

This setup gives you a professional development environment for your RAG chatbot with debugging, testing, and deployment capabilities all within VS Code!