**1) Component map → which RAG piece does what**

* data/ — raw docs (collect data).
* ingest/
  + chunker.py — preprocess & chunk documents.
  + loader.py — load PDF/TXT files into Document objects.
  + embedder.py — create embeddings (SentenceTransformerEmbeddings).
  + indexer.py — build and persist Chroma vector DB.
* runtime/
  + retriever.py — query vector DB, return top-k chunks.
  + prompt.py — assemble prompt / formatting.
  + generator.py — LLM call (OpenAI option + local fallback).
* ui/
  + app.py — Streamlit UI and orchestration (start ingest, query, show results).
* config/
  + .env or config constants.

**2) Project layout (copy this)**

rag-streamlit/

├─ data/ # put your PDFs/TXTs here

├─ ingest/

│ ├─ \_\_init\_\_.py

│ ├─ loader.py

│ ├─ chunker.py

│ ├─ embedder.py

│ └─ indexer.py

├─ runtime/

│ ├─ \_\_init\_\_.py

│ ├─ retriever.py

│ ├─ prompt.py

│ └─ generator.py

├─ ui/

│ ├─ \_\_init\_\_.py

│ └─ app.py

├─ .env.example

├─ requirements.txt

├─ Dockerfile

└─ README.md

**3) Full code (drop each file into the structure above)**

Implementation notes before code:

* I keep your Chroma + SentenceTransformer choices.
* generator.py provides an **OpenAI** path if you set OPENAI\_API\_KEY and a simple **fallback** that returns a grounded excerpt (safe, lightweight).
* This is intentionally minimal so it runs locally without huge model downloads. If you want an actual local HF generation pipeline, I can add it as a next step.

**requirements.txt**

streamlit

langchain

langchain-community

sentence-transformers

chromadb

python-dotenv

openai

tqdm

PyPDF2

Note: langchain-community and chromadb names may vary in versions — install the versions that match your environment.

**.env.example**

# Optional if you will use OpenAI for generation

OPENAI\_API\_KEY=

LLM\_PROVIDER=openai # or "local" for fallback behaviour

PERSIST\_DIRECTORY=./today

DATA\_DIRECTORY=./data

COLLECTION\_NAME=testing1

EMBED\_MODEL=all-MiniLM-L6-v2

**ingest/loader.py**

# ingest/loader.py

from langchain\_community.document\_loaders import DirectoryLoader, TextLoader, PyPDFLoader

from pathlib import Path

from typing import List

from langchain.schema import Document

DATA\_DIRECTORY = Path(\_\_file\_\_).resolve().parents[1] / "data"

def load\_documents(data\_dir: str = None) -> List[Document]:

"""

Load PDFs and TXTs from the data directory and return a list of LangChain Documents.

"""

base = DATA\_DIRECTORY if data\_dir is None else Path(data\_dir)

docs = []

# PDFs

pdf\_loader = DirectoryLoader(str(base), glob="\*\*/\*.pdf", loader\_cls=PyPDFLoader, silent\_errors=True)

docs.extend(pdf\_loader.load())

# TXTs

txt\_loader = DirectoryLoader(str(base), glob="\*\*/\*.txt", loader\_cls=TextLoader, silent\_errors=True)

docs.extend(txt\_loader.load())

return docs

**ingest/chunker.py**

# ingest/chunker.py

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from typing import List

from langchain.schema import Document

def chunk\_documents(docs: List[Document], chunk\_size=1000, chunk\_overlap=200) -> List[Document]:

"""

Use RecursiveCharacterTextSplitter to split documents into chunks (keeps metadata).

"""

splitter = RecursiveCharacterTextSplitter(chunk\_size=chunk\_size, chunk\_overlap=chunk\_overlap)

chunks = splitter.split\_documents(docs)

return chunks

**ingest/embedder.py**

# ingest/embedder.py

import os

from langchain\_community.embeddings import SentenceTransformerEmbeddings

def get\_embeddings(model\_name: str = None):

"""

Return a SentenceTransformerEmbeddings instance (wraps HuggingFace sentence-transformers).

"""

model\_name = model\_name or os.getenv("EMBED\_MODEL", "all-MiniLM-L6-v2")

emb = SentenceTransformerEmbeddings(model\_name=model\_name)

return emb

**ingest/indexer.py**

# ingest/indexer.py

import os

from langchain\_community.vectorstores import Chroma

from typing import List

from langchain.schema import Document

def build\_and\_persist\_vectorstore(chunks: List[Document], embedding\_function, persist\_directory: str, collection\_name: str):

"""

Create Chroma vectorstore from chunked documents and persist to disk.

"""

# simple safety check

if not chunks:

raise ValueError("No chunks to index.")

# build Chroma from documents

vectordb = Chroma.from\_documents(

documents=chunks,

embedding=embedding\_function,

persist\_directory=persist\_directory,

collection\_name=collection\_name

)

vectordb.persist()

return vectordb

**runtime/retriever.py**

# runtime/retriever.py

from langchain\_community.vectorstores import Chroma

from langchain\_community.embeddings import SentenceTransformerEmbeddings

import os

def load\_vectorstore(persist\_directory: str, collection\_name: str, model\_name: str = None):

"""

Load existing Chroma vectorstore from disk. Returns Chroma instance or None.

"""

model\_name = model\_name or os.getenv("EMBED\_MODEL", "all-MiniLM-L6-v2")

emb = SentenceTransformerEmbeddings(model\_name=model\_name)

if not os.path.exists(persist\_directory) or not any(os.scandir(persist\_directory)):

return None

return Chroma(persist\_directory=persist\_directory, embedding\_function=emb, collection\_name=collection\_name)

def retrieve(query: str, vectordb, k: int = 3):

"""

Simple semantic search wrapper.

Returns list of Documents (langchain schema objects).

"""

if vectordb is None:

return []

docs = vectordb.similarity\_search(query, k=k)

return docs

**runtime/prompt.py**

# runtime/prompt.py

def build\_context\_from\_docs(docs, max\_chars=3000):

"""

Join top-k docs into a single context string limited to max\_chars.

Keeps a small source header for traceability.

"""

parts = []

for d in docs:

meta = getattr(d, "metadata", {}) or {}

src = meta.get("source") or meta.get("doc\_id") or "unknown"

header = f"--- Source: {src} ---"

content = d.page\_content if hasattr(d, "page\_content") else (d.content if hasattr(d, "content") else str(d))

parts.append(f"{header}\n{content}")

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

if len(context) > max\_chars:

context = context[:max\_chars] + "\n\n...[truncated]"

return context

BASE\_PROMPT = """You are a helpful assistant. Use the following context to answer the question.

If the context does not contain the answer, say "I don't know".

Context:

{context}

Question:

{question}

Answer:

"""

def build\_prompt(docs, question):

ctx = build\_context\_from\_docs(docs)

return BASE\_PROMPT.format(context=ctx, question=question)

**runtime/generator.py**

# runtime/generator.py

import os

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

LLM\_PROVIDER = os.getenv("LLM\_PROVIDER", "openai")

def generate\_with\_openai(prompt: str, max\_tokens: int = 256, temperature: float = 0.2):

import openai

openai.api\_key = OPENAI\_KEY

# use ChatCompletion for chat models

model = os.getenv("LLM\_MODEL", "gpt-3.5-turbo")

try:

resp = openai.ChatCompletion.create(

model=model,

messages=[{"role": "user", "content": prompt}],

temperature=temperature,

max\_tokens=max\_tokens

)

return resp["choices"][0]["message"]["content"].strip()

except Exception as e:

return f"[OpenAI generation error] {e}"

def generate\_fallback(prompt: str):

"""

Lightweight fallback: return a grounded summary by echoing the context header + first bytes.

Not a replacement for an LLM, but safe and fast.

"""

# Try to extract the "Context:" section from the prompt

try:

ctx = prompt.split("Context:")[1].split("Question:")[0].strip()

excerpt = ctx[:1500]

return "Grounded excerpt from retrieved docs:\n\n" + excerpt + ("\n\n...[truncated]" if len(ctx) > 1500 else "")

except Exception:

return "Could not generate a fallback answer."

def generate(prompt: str, \*\*kwargs):

if LLM\_PROVIDER == "openai" and OPENAI\_KEY:

return generate\_with\_openai(prompt, \*\*kwargs)

else:

return generate\_fallback(prompt)

**ui/app.py (Streamlit frontend, refactored)**

# ui/app.py

import streamlit as st

from dotenv import load\_dotenv

import os

from ingest.loader import load\_documents

from ingest.chunker import chunk\_documents

from ingest.embedder import get\_embeddings

from ingest.indexer import build\_and\_persist\_vectorstore

from runtime.retriever import load\_vectorstore, retrieve

from runtime.prompt import build\_prompt

from runtime.generator import generate

load\_dotenv() # load .env if present

# Config (use env or defaults)

DATA\_DIRECTORY = os.getenv("DATA\_DIRECTORY", "./data")

PERSIST\_DIRECTORY = os.getenv("PERSIST\_DIRECTORY", "./today")

COLLECTION\_NAME = os.getenv("COLLECTION\_NAME", "testing1")

EMBED\_MODEL = os.getenv("EMBED\_MODEL", "all-MiniLM-L6-v2")

st.set\_page\_config(page\_title="RAG Streamlit App", layout="wide")

st.title("RAG App — modular version")

st.markdown("Use the \*\*Ingest\*\* button if you added/changed files in the `data/` folder.")

# Session-state vectorstore

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

st.session\_state.vectordb = None

# Ingest pipeline UI

with st.expander("1) Ingest documents (click to run)"):

st.write(f"Data directory: `{DATA\_DIRECTORY}`")

if st.button("Ingest Documents"):

with st.spinner("Loading documents..."):

docs = load\_documents(DATA\_DIRECTORY)

st.write(f"Loaded {len(docs)} documents.")

if docs:

chunks = chunk\_documents(docs)

st.write(f"Created {len(chunks)} chunks.")

emb = get\_embeddings(EMBED\_MODEL)

st.write("Embedding model ready.")

try:

vectordb = build\_and\_persist\_vectorstore(chunks, emb, PERSIST\_DIRECTORY, COLLECTION\_NAME)

st.success("Vectorstore built and persisted.")

st.session\_state.vectordb = vectordb

except Exception as e:

st.error(f"Failed to build index: {e}")

else:

st.warning("No documents found. Place PDFs / TXTs into the data directory.")

# Load existing store if not loaded

if st.session\_state.vectordb is None:

try:

vect = load\_vectorstore(PERSIST\_DIRECTORY, COLLECTION\_NAME, model\_name=EMBED\_MODEL)

if vect is not None:

st.session\_state.vectordb = vect

st.success("Loaded existing vectorstore from disk.")

else:

st.info("No persisted vectorstore found. Please ingest documents.")

except Exception as e:

st.error(f"Error loading persisted vectorstore: {e}")

# Query UI

st.markdown("---")

st.header("2) Ask a question")

col1, col2 = st.columns([4, 1])

with col1:

question = st.text\_input("Enter your question here")

with col2:

k = st.number\_input("Top-k", min\_value=1, max\_value=10, value=3, step=1)

if st.button("Get Answer"):

if not st.session\_state.vectordb:

st.warning("Vectorstore not ready. Ingest documents first.")

elif not question.strip():

st.warning("Please enter a question.")

else:

with st.spinner("Retrieving documents..."):

docs = retrieve(question, st.session\_state.vectordb, k=k)

if not docs:

st.info("No relevant documents found.")

else:

prompt = build\_prompt(docs, question)

with st.spinner("Calling generator..."):

answer = generate(prompt, max\_tokens=300, temperature=0.2)

st.subheader("Answer")

st.info(answer)

with st.expander("Retrieved documents (for traceability)"):

for i, d in enumerate(docs):

meta = getattr(d, "metadata", {}) or {}

src = meta.get("source", meta.get("doc\_id", "unknown"))

st.write(f"--- [{i+1}] Source: {src}")

content = d.page\_content if hasattr(d, "page\_content") else getattr(d, "content", str(d))

st.text(content[:1000] + ("...[truncated]" if len(content) > 1000 else ""))

st.markdown("---")

st.markdown("Small note: currently generation uses OpenAI (if configured) or a safe fallback that returns grounded excerpts.")

**4) How to run locally (step-by-step)**

1. Create the project directory and files shown above. Put PDFs/TXTs into rag-streamlit/data/.
2. Create virtual environment and install:

python -m venv .venv

source .venv/bin/activate

pip install --upgrade pip

pip install -r requirements.txt

1. Copy .env.example to .env and populate if you want OpenAI:

cp .env.example .env

# edit .env to set OPENAI\_API\_KEY if you want LLM answers

1. Start Streamlit:

streamlit run ui/app.py

Open <http://localhost:8501>

1. Workflow:

* Click **Ingest Documents** (this loads & chunks & builds the Chroma DB in PERSIST\_DIRECTORY).
* Ask questions — top-k results will be retrieved and you’ll see an LLM answer (if OpenAI configured) or a grounded excerpt fallback.

**5) Dockerize (simple Streamlit container)**

**Dockerfile**

FROM python:3.11-slim

WORKDIR /app

COPY . /app

# system packages needed for sentence-transformers/chromadb/pdf read

RUN apt-get update && apt-get install -y build-essential git ffmpeg libsndfile1 poppler-utils && rm -rf /var/lib/apt/lists/\*

RUN pip install --upgrade pip

RUN pip install -r requirements.txt

ENV PYTHONUNBUFFERED=1

ENV STREAMLIT\_SERVER\_ENABLE\_CORS=false

EXPOSE 8501

CMD ["streamlit", "run", "ui/app.py", "--server.port=8501", "--server.address=0.0.0.0"]

**Build & run**

docker build -t rag-streamlit .

docker run --rm -p 8501:8501 -e OPENAI\_API\_KEY=yourkey -v $(pwd)/data:/app/data rag-streamlit