**RAG based chatbot system**

<https://colab.research.google.com/drive/1pat55z_iiLqzInsLi3sWS2wekFCXprQW?usp=sharing>

full project - <https://colab.research.google.com/drive/1O7cdBtiP_GNXgL9Iz4LPzYfvTKtMtv25?usp=sharing>

Absolutely! Below is a **complete, runnable Streamlit RAG chatbot** for a realistic scenario:

**Scenario**: An **IT Support & Customer Success chatbot** for a B2B SaaS company.  
It answers questions from product manuals, onboarding guides, release notes, and policy docs, returning **concise answers with citations**.

You’ll get:

* requirements.txt
* ingest.py → loads files, chunks, embeds, builds retrievers (Chroma + BM25), and persists.
* app.py → Streamlit UI, chat history, **EnsembleRetriever** (semantic + keyword), optional **ContextualCompression** step, and nice **source citations**.
* Works locally with **Chroma** (vector DB) and **OpenAI embeddings/LLM** (swap to your preferred providers if you like).

**1) requirements.txt**

streamlit>=1.33

langchain>=0.2.14

langchain-community>=0.2.12

langchain-openai>=0.1.21

langchain-text-splitters>=0.2.2

langchain-chroma>=0.1.2

chromadb>=0.5.5

rank-bm25>=0.2.2

pydantic>=1.10,<2

tiktoken>=0.7.0

python-dotenv>=1.0.1

**2) Project structure**

rag-support-bot/

├─ requirements.txt

├─ .env

├─ data/ # put your PDFs / TXTs / MDs here

│ ├─ onboarding.md

│ ├─ release\_notes\_Q2.pdf

│ └─ product\_manual.txt

├─ ingest.py

└─ app.py

**.env** (create it):

OPENAI\_API\_KEY=sk-your-key

**3) ingest.py (build the RAG index)**

# ingest.py

import os

from pathlib import Path

from dotenv import load\_dotenv

from langchain\_community.document\_loaders import (

TextLoader,

PyPDFLoader,

UnstructuredMarkdownLoader,

)

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

from langchain\_openai import OpenAIEmbeddings

from langchain\_chroma import Chroma

from langchain.schema import Document

from langchain\_community.retrievers import BM25Retriever # keyword retriever

load\_dotenv()

DATA\_DIR = Path("data")

PERSIST\_DIR = "chroma\_store"

COLLECTION\_NAME = "support\_kb"

def load\_all\_documents(data\_dir: Path) -> list[Document]:

docs: list[Document] = []

for p in data\_dir.glob("\*\*/\*"):

if p.is\_dir():

continue

ext = p.suffix.lower()

try:

if ext in [".txt", ".log"]:

loader = TextLoader(str(p), encoding="utf-8")

elif ext in [".md", ".markdown"]:

loader = UnstructuredMarkdownLoader(str(p))

elif ext in [".pdf"]:

loader = PyPDFLoader(str(p))

else:

print(f"Skipping unsupported file: {p}")

continue

docs.extend(loader.load())

print(f"Loaded: {p}")

except Exception as e:

print(f"Failed to load {p}: {e}")

return docs

def split\_docs(docs: list[Document]) -> list[Document]:

splitter = RecursiveCharacterTextSplitter(

chunk\_size=800, chunk\_overlap=150,

separators=["\n\n", "\n", " ", ""]

)

return splitter.split\_documents(docs)

def main():

# 1) Load raw files

raw\_docs = load\_all\_documents(DATA\_DIR)

if not raw\_docs:

raise RuntimeError(

f"No documents found in {DATA\_DIR}. Add PDFs/TXT/MD files and re-run."

)

# 2) Chunking

chunks = split\_docs(raw\_docs)

print(f"Total chunks: {len(chunks)}")

# 3) Embeddings + Vector DB

embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

vectordb = Chroma(

collection\_name=COLLECTION\_NAME,

persist\_directory=PERSIST\_DIR,

embedding\_function=embeddings,

)

# 4) Upsert (simple: add all)

# If you re-run often, you might want to wipe first:

# vectordb.delete\_collection() # uncomment for a full rebuild

vectordb.add\_documents(chunks)

vectordb.persist()

print(f"✅ Chroma persisted to: {PERSIST\_DIR}")

# 5) Build and persist a BM25 index (for hybrid retrieval in app.py)

# We'll dump the raw text corpus to a simple pickle for quick loading.

# BM25Retriever is in-memory; we recreate it in app.py from text.

text\_corpus = [d.page\_content for d in chunks]

corpus\_path = Path(PERSIST\_DIR) / "bm25\_corpus.txt"

corpus\_path.parent.mkdir(parents=True, exist\_ok=True)

with open(corpus\_path, "w", encoding="utf-8") as f:

for t in text\_corpus:

# Separate docs with a \n---\n marker for easy reload

f.write(t.replace("\n", " ").strip() + "\n---\n")

print(f"✅ BM25 corpus saved: {corpus\_path}")

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

main()

**4) app.py (Streamlit chatbot with RAG)**

# app.py

import os

import time

from pathlib import Path

from typing import List

import streamlit as st

from dotenv import load\_dotenv

from langchain\_openai import ChatOpenAI, OpenAIEmbeddings

from langchain\_chroma import Chroma

from langchain.schema import Document

from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder

from langchain\_core.runnables import (

RunnableParallel, RunnableSequence, RunnableLambda, RunnablePassthrough

)

from langchain\_core.output\_parsers import StrOutputParser

# Retrievers

from langchain\_community.retrievers import BM25Retriever

from langchain.retrievers import EnsembleRetriever, ContextualCompressionRetriever

from langchain.retrievers.document\_compressors import LLMChainExtractor

# --------------------------

# App config

# --------------------------

load\_dotenv()

st.set\_page\_config(page\_title="SaaS Support RAG Bot", page\_icon="🛠️", layout="wide")

st.title("🛠️ SaaS Support RAG Bot")

st.caption("Ask about product features, onboarding steps, release notes, and policies. Answers include citations.")

# --------------------------

# Constants / Paths

# --------------------------

PERSIST\_DIR = "chroma\_store"

COLLECTION\_NAME = "support\_kb"

BM25\_CORPUS = Path(PERSIST\_DIR) / "bm25\_corpus.txt"

# --------------------------

# Sidebar: Settings

# --------------------------

with st.sidebar:

st.header("Settings")

model\_name = st.selectbox("LLM", ["gpt-4o-mini", "gpt-4o"], index=0)

temperature = st.slider("Temperature", 0.0, 1.0, 0.0, 0.1)

top\_k = st.slider("Top K (base retrieval)", 1, 10, 4)

use\_compression = st.checkbox("Use contextual compression (re-rank/trim with LLM)", value=True)

hybrid\_weight = st.slider("Hybrid weight (semantic vs keyword)", 0.0, 1.0, 0.6, 0.05)

st.markdown("---")

st.markdown("\*\*Tip\*\*: Toggle compression if responses seem too long or noisy.")

# --------------------------

# Init LLM + Vector DB

# --------------------------

@st.cache\_resource(show\_spinner=False)

def get\_llm(name: str, temp: float):

return ChatOpenAI(model=name, temperature=temp, streaming=False)

@st.cache\_resource(show\_spinner=False)

def get\_vectordb():

embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

return Chroma(

collection\_name=COLLECTION\_NAME,

persist\_directory=PERSIST\_DIR,

embedding\_function=embeddings,

)

@st.cache\_resource(show\_spinner=False)

def get\_bm25\_retriever() -> BM25Retriever:

# Rehydrate the BM25 corpus saved in ingest.py

if not BM25\_CORPUS.exists():

raise RuntimeError("BM25 corpus missing. Run ingest.py first.")

texts: List[str] = []

buff = []

with open(BM25\_CORPUS, "r", encoding="utf-8") as f:

for line in f:

if line.strip() == "---":

if buff:

texts.append(" ".join(buff).strip())

buff = []

else:

buff.append(line.strip())

if buff:

texts.append(" ".join(buff).strip())

# Create Documents for BM25

docs = [Document(page\_content=t) for t in texts if t]

retriever = BM25Retriever.from\_documents(docs)

retriever.k = 6

return retriever

llm = get\_llm(model\_name, temperature)

vectordb = get\_vectordb()

bm25 = get\_bm25\_retriever()

# Base retrievers

vector\_retriever = vectordb.as\_retriever(

search\_type="similarity",

search\_kwargs={"k": top\_k},

)

# Ensemble (hybrid: semantic + keyword)

ensemble\_retriever = EnsembleRetriever(

retrievers=[vector\_retriever, bm25],

weights=[hybrid\_weight, 1.0 - hybrid\_weight],

)

# Optional compression (LLM trims to only the most relevant snippets)

if use\_compression:

compressor = LLMChainExtractor.from\_llm(llm)

retriever = ContextualCompressionRetriever(

base\_retriever=ensemble\_retriever, base\_compressor=compressor

)

else:

retriever = ensemble\_retriever

# --------------------------

# System + Prompt

# --------------------------

SYSTEM = (

"You are a helpful IT Support & Customer Success assistant for a B2B SaaS platform. "

"Answer clearly and concisely using only the provided context. "

"If the answer isn't in the context, say you don't know. "

"Cite sources as [S1], [S2] matching the provided documents."

)

RAG\_PROMPT = ChatPromptTemplate.from\_messages(

[

("system", SYSTEM),

MessagesPlaceholder(variable\_name="chat\_history"),

(

"human",

"User question: {question}\n\n"

"Context (top documents):\n{context}\n\n"

"Return a concise answer with citations like [S1], [S2]."

),

]

)

# --------------------------

# Utilities

# --------------------------

def docs\_to\_context(docs: List[Document]) -> str:

parts = []

for i, d in enumerate(docs, start=1):

src = d.metadata.get("source", "unknown")

parts.append(f"[S{i}] ({src}) {d.page\_content}")

return "\n\n".join(parts)

def format\_citations(docs: List[Document]) -> str:

lines = []

for i, d in enumerate(docs, start=1):

src = d.metadata.get("source", "unknown")

lines.append(f"[S{i}] {src}")

return "\n".join(lines) if lines else "No sources."

# Chain: retrieve -> prompt -> llm -> text

def make\_chain():

retrieve = RunnableLambda(lambda q: retriever.get\_relevant\_documents(q))

to\_context = RunnableLambda(lambda docs: {

"docs": docs,

"context": docs\_to\_context(docs)

})

# Build prompt+LLM pipeline

gen = RunnableSequence(

RAG\_PROMPT,

llm,

StrOutputParser()

)

# Fan-in final result with answer + docs (for citations)

final = RunnableParallel({

"answer": gen,

"docs": RunnablePassthrough()

})

# Full chain from user input

# input -> retrieve -> to\_context -> final(answer+docs)

chain = RunnableSequence(

retrieve,

to\_context,

RunnableLambda(lambda d: {"context": d["context"], "docs": d["docs"]}),

RunnableLambda(lambda d, \*\*kwargs: {

"question": kwargs.get("\_\_input\_\_"), # original user question

"context": d["context"], # formatted context

"docs": d["docs"]

}),

final

)

return chain

chain = make\_chain()

# --------------------------

# Session state for chat

# --------------------------

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

st.session\_state.history = [] # list of {"role": "user"/"ai", "content": str}

# Display history

for msg in st.session\_state.history:

if msg["role"] == "user":

st.chat\_message("user").markdown(msg["content"])

else:

st.chat\_message("assistant").markdown(msg["content"])

# --------------------------

# Chat input

# --------------------------

user\_input = st.chat\_input("Ask about setup, SSO, roles, rate limits, releases, etc.")

if user\_input:

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

st.chat\_message("user").markdown(user\_input)

# Build a chat\_history string for the prompt (lightweight)

chat\_history\_text = "\n".join(

[f"{m['role'].upper()}: {m['content']}" for m in st.session\_state.history[-6:]]

)

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

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

# Invoke chain → returns {"answer": "...", "docs": [...]}

result = chain.invoke({"question": user\_input, "chat\_history": chat\_history\_text})

answer = result["answer"]

docs = result["docs"]

st.markdown(answer)

st.markdown("\*\*Sources\*\*")

st.code(format\_citations(docs))

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

**5) How to run**

pip install -r requirements.txt

# Put your PDFs/TXTs/MDs in ./data

python ingest.py

streamlit run app.py

Open the URL Streamlit prints (usually <http://localhost:8501>), ask questions like:

* “How do I enable SSO (SAML)?”
* “What changed in the Q2 release?”
* “What are the default rate limits per API key?”
* “How do I invite team members and assign roles?”

**What this demonstrates (RAG components)**

* **Loaders**: PDF/TXT/MD.
* **Chunking**: RecursiveCharacterTextSplitter.
* **Embeddings + Vector DB**: OpenAI + Chroma (persistent).
* **Retrievers**:
  + **Vector similarity** (Chroma.as\_retriever).
  + **Keyword BM25** (BM25Retriever).
  + **Hybrid** via **EnsembleRetriever** (weighted fusion).
  + **Optional Contextual Compression** (LLM trims irrelevant parts).
* **Prompting**: ChatPromptTemplate with **chat history** & **context injection**.
* **Chain**: Runnable graph (retrieve → format → LLM → merge → display).
* **Citations**: Clear sources [S1], [S2] from retrieved docs.
* **Streamlit UI**: simple chat-like experience.

**Notes & Extensions**

* Swap OpenAIEmbeddings/ChatOpenAI with other providers (Azure OpenAI, Anthropic, local LLM) if needed.
* Add **metadata filters** (e.g., only “policy” docs) by tagging metadata during ingestion and using retriever search\_kwargs.
* Add **Self-Query Retriever** for natural-language filters (date ranges, categories).
* Add a **feedback button** to store bad answers and improve your dataset.
* Implement **update/delete** flows by calling vectordb.delete() and re-adding documents.

If you want, I can extend this with **Self-Query Retriever** (metadata-aware natural filters) and/or **re-ranking** (e.g., Cohere Rerank) as an extra step!

We’ll imagine a **Customer Support Chatbot** for an **E-commerce company (say Flipkart/Amazon-like)**.  
The bot should:

* Answer FAQs (shipping, returns, payments, etc.)
* Search knowledge base docs (policies, manuals).
* Use **RAG (Retrieval-Augmented Generation)** to pull facts from a **vector database (ChromaDB in our case)**.
* Have a simple **Streamlit UI**.

**🔹 Module-wise Explanation**

**1. ingest.py (Data Loading + Preprocessing + Vector DB storage)**

This module handles **loading documents, splitting them, embedding them, and storing into Chroma vector DB**.

# ingest.py

from langchain\_community.document\_loaders import TextLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain\_openai import OpenAIEmbeddings

from langchain\_chroma import Chroma

def ingest\_docs():

# Step 1: Load company knowledge base (FAQs, policies, etc.)

loader = TextLoader("data/company\_faq.txt")

documents = loader.load()

# Step 2: Split into smaller chunks for better retrieval

splitter = RecursiveCharacterTextSplitter(chunk\_size=500, chunk\_overlap=50)

chunks = splitter.split\_documents(documents)

# Step 3: Create embeddings for semantic similarity

embeddings = OpenAIEmbeddings()

# Step 4: Store in Chroma (Vector Store)

vectordb = Chroma.from\_documents(chunks, embeddings, persist\_directory="db/chroma\_db")

vectordb.persist()

print("✅ Data Ingested and Stored in Vector DB")

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

ingest\_docs()

**Uses**

* Run this script once to prepare data.
* Example: company\_faq.txt might include policies like "Return period is 7 days for electronics".

**Pros**: Modular, reusable, scalable.  
**Cons**: Need to rerun ingestion when knowledge base updates.

**2. retriever.py (Retriever Configurations)**

This defines **different retrievers** we can use with our vector DB.

# retriever.py

from langchain\_chroma import Chroma

from langchain\_openai import OpenAIEmbeddings

def get\_retrievers():

embeddings = OpenAIEmbeddings()

vectordb = Chroma(persist\_directory="db/chroma\_db", embedding\_function=embeddings)

retrievers = {

"similarity": vectordb.as\_retriever(search\_kwargs={"k": 3}),

"mmr": vectordb.as\_retriever(search\_type="mmr", search\_kwargs={"k": 3, "lambda\_mult": 0.5}),

"metadata": vectordb.as\_retriever(search\_kwargs={"k": 3, "filter": {"category": "returns"}}),

}

return retrievers

**Uses**

* Choose retriever dynamically.
* Example: If a user says “what is return policy?”, the **metadata retriever** fetches only return-related docs.

**3. rag\_chain.py (RAG Pipeline Construction)**

This builds the **Retrieval-Augmented Generation chain** with an LLM + retriever.

# rag\_chain.py

from langchain.chains import RetrievalQA

from langchain\_openai import ChatOpenAI

from retriever import get\_retrievers

def get\_rag\_chain(retriever\_type="similarity"):

retrievers = get\_retrievers()

retriever = retrievers[retriever\_type]

llm = ChatOpenAI(model="gpt-4", temperature=0)

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

llm=llm,

retriever=retriever,

chain\_type="stuff"

)

return qa\_chain

**Uses**

* Ties together LLM + retriever.
* Example: “Can I return a phone after 10 days?” → retriever fetches return policy → LLM explains politely.

**4. app.py (Streamlit UI + Chat Loop)**

This is the **user-facing chatbot app**.

# app.py

import streamlit as st

from rag\_chain import get\_rag\_chain

st.set\_page\_config(page\_title="E-commerce Chatbot", layout="wide")

st.title("🛍️ E-commerce Customer Support Chatbot")

st.write("Ask me anything about orders, returns, shipping, and payments!")

# Sidebar to pick retriever type

retriever\_type = st.sidebar.selectbox("Select Retrieval Mode", ["similarity", "mmr", "metadata"])

qa\_chain = get\_rag\_chain(retriever\_type)

# Session state for storing conversation

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

st.session\_state["chat\_history"] = []

# User input

query = st.text\_input("💬 Your Question:")

if query:

response = qa\_chain.run(query)

st.session\_state["chat\_history"].append((query, response))

# Display chat history

for q, r in st.session\_state["chat\_history"]:

st.markdown(f"\*\*You:\*\* {q}")

st.markdown(f"\*\*Bot:\*\* {r}")

st.write("---")

**Uses**

* Provides **UI** for interacting.
* Switch retriever type from sidebar.
* Conversation context stored with session\_state.

**5. Directory Structure**

chatbot/

│── data/

│ └── company\_faq.txt

│── db/

│ └── chroma\_db/ # Vector store persistence

│── ingest.py

│── retriever.py

│── rag\_chain.py

│── app.py

**🔹 Real-World Company Scenario**

* **Flipkart** deploys this chatbot on their help page.
* A user types: *“Can I return a laptop after 15 days?”*
  1. Query sent → retriever fetches "electronics return policy (7 days)".
  2. LLM generates: *“Laptops can be returned only within 7 days. Since it’s 15 days, the return window has closed.”*
* **Business value**: Reduces load on support staff, 24x7 automated help.

✅ **Summary of Modules**

* **ingest.py** → Load + embed + store docs in Chroma.
* **retriever.py** → Different retrievers (similarity, mmr, metadata).
* **rag\_chain.py** → Build RetrievalQA chain.
* **app.py** → Streamlit chatbot UI.

Would you like me to **extend this** by showing how we can add **hybrid search retriever (keyword + vector)** with Weaviate or Pinecone (like in enterprise-grade systems)?

**Project 2**

let’s build a **production-style RAG app** for a different, very real IT scenario and ship it with a Streamlit UI + deployables.

**🏢 Scenario: “DevOps Copilot” for a Cloud/SRE Team**

Your company runs multi-tenant Kubernetes clusters and cloud infra (AWS/GCP). Engineers need a copilot that can answer questions from:

* **Runbooks** (incident procedures, on-call checklists)
* **Terraform modules** (variables, examples, provider versions)
* **Postmortems** (lessons learned)
* **Internal wiki** (service ownership, SLOs, escalation paths)
* **Changelogs / Release notes** (what changed last week)

We’ll support:

* **Hybrid retrieval** (BM25 keyword + vector similarity)
* **Contextual compression** (LLM trims noisy chunks)
* **Metadata filtering** (env/service/team/confidentiality)
* **(Optional) Self-Query Retriever** (LLM turns NL filters → metadata filters)
* **Citations** and **feedback capture**
* **Dockerized** for prod

**📁 Project Layout**

devops-copilot/

├─ requirements.txt

├─ .env.example

├─ Dockerfile

├─ README.md

├─ data/ # put your MD/TXT/PDF here (runbooks, TF modules, etc.)

│ ├─ runbooks/oncall\_redis.md

│ ├─ tf/eks\_module.md

│ ├─ wiki/slo\_p95.md

│ └─ incidents/2025-08-01-redis-outage.md

├─ store/ # (auto-created) chroma persistence + bm25 corpus

├─ ingest.py

└─ app.py

**📦 requirements.txt**

streamlit>=1.34

python-dotenv>=1.0.1

pydantic>=1.10,<2

tiktoken>=0.7.0

langchain>=0.2.14

langchain-community>=0.2.12

langchain-openai>=0.1.21

langchain-text-splitters>=0.2.2

langchain-chroma>=0.1.2

chromadb>=0.5.5

rank-bm25>=0.2.2

PyYAML>=6.0.1

**🔑 .env.example**

# copy to .env and fill

OPENAI\_API\_KEY=sk-...

# Optional: simple Streamlit auth token (demo-level)

APP\_AUTH\_TOKEN=letmein

**🧠 ingest.py (load → chunk → embed → persist + BM25 corpus)**

# ingest.py

import os

from pathlib import Path

from typing import Dict, Any, List, Optional

import re

import yaml

from dotenv import load\_dotenv

from langchain.schema import Document

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

from langchain\_openai import OpenAIEmbeddings

from langchain\_chroma import Chroma

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

load\_dotenv()

DATA\_DIR = Path("data")

PERSIST\_DIR = "store/chroma"

COLL = "devops\_kb"

BM25\_CORPUS\_PATH = Path("store") / "bm25\_corpus.ndjson"

SUPPORTED = {".md", ".txt", ".pdf"}

FRONT\_MATTER\_RE = re.compile(r"^---\s\*\n(.\*?)\n---\s\*\n", re.DOTALL)

def parse\_front\_matter(text: str) -> (Dict[str, Any], str):

"""Extract YAML front matter if present (--- ... ---)."""

m = FRONT\_MATTER\_RE.match(text)

if not m:

return {}, text

meta\_raw = m.group(1)

rest = text[m.end():]

try:

meta = yaml.safe\_load(meta\_raw) or {}

if not isinstance(meta, dict):

meta = {}

except Exception:

meta = {}

return meta, rest

def load\_docs(data\_dir: Path) -> List[Document]:

docs: List[Document] = []

for p in data\_dir.rglob("\*"):

if p.is\_dir() or p.suffix.lower() not in SUPPORTED:

continue

try:

if p.suffix.lower() in {".txt", ".md"}:

raw = p.read\_text(encoding="utf-8", errors="ignore")

front, body = parse\_front\_matter(raw)

# Fall back to loader behavior for page breaks etc. (simple here)

content = body.strip()

meta = {

"source": str(p),

\*\*front,

# Derive a few metadata hints from path

"team": front.get("team") or p.parts[-2] if len(p.parts) > 1 else "unknown",

"service": front.get("service") or ("eks" if "eks" in p.stem.lower() else "unknown"),

"env": front.get("env") or ("prod" if "prod" in str(p).lower() else "unknown"),

"confidentiality": front.get("confidentiality", "internal"),

}

docs.append(Document(page\_content=content, metadata=meta))

elif p.suffix.lower() == ".pdf":

for d in PyPDFLoader(str(p)).load():

# Attach derived metadata too

d.metadata.update({

"source": str(p),

"team": d.metadata.get("team", "unknown"),

"service": d.metadata.get("service", "unknown"),

"env": ("prod" if "prod" in str(p).lower() else "unknown"),

"confidentiality": d.metadata.get("confidentiality", "internal"),

})

docs.append(d)

print(f"Loaded {p}")

except Exception as e:

print(f"⚠️ Failed to load {p}: {e}")

return docs

def chunk(docs: List[Document]) -> List[Document]:

splitter = RecursiveCharacterTextSplitter(

chunk\_size=900, chunk\_overlap=150,

separators=["\n\n", "\n", " ", ""]

)

return splitter.split\_documents(docs)

def write\_bm25\_corpus(chunks: List[Document], path: Path) -> None:

"""Store as ndjson lines: {'text': ..., 'metadata': {...}}."""

import json

path.parent.mkdir(parents=True, exist\_ok=True)

with path.open("w", encoding="utf-8") as f:

for d in chunks:

f.write(json.dumps({"text": d.page\_content, "metadata": d.metadata}, ensure\_ascii=False) + "\n")

print(f"✅ BM25 corpus saved to {path}")

def main():

raw = load\_docs(DATA\_DIR)

if not raw:

raise RuntimeError("No documents found in ./data. Add runbooks/wiki/TF docs and retry.")

chunks = chunk(raw)

print(f"Chunks: {len(chunks)}")

embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

vectordb = Chroma(

collection\_name=COLL,

persist\_directory=PERSIST\_DIR,

embedding\_function=embeddings,

)

# Simple rebuild: wipe + add (comment out wipe if you want incremental)

try:

vectordb.delete\_collection()

except Exception:

pass

vectordb.add\_documents(chunks)

vectordb.persist()

print(f"✅ Chroma persisted at {PERSIST\_DIR}")

write\_bm25\_corpus(chunks, BM25\_CORPUS\_PATH)

print("✅ Ingestion complete.")

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

main()

**Tip:** Add YAML front-matter to docs to improve metadata filtering:

---

team: sre

service: redis

env: prod

confidentiality: internal

updated\_at: 2025-08-01

---

# Redis On-Call Runbook

...

**💬 app.py (Streamlit UI + RAG chain + hybrid retrieval + compression + filters + citations + feedback)**

# app.py

import os, json, time

from pathlib import Path

from typing import List, Dict, Any

import streamlit as st

from dotenv import load\_dotenv

from langchain\_openai import ChatOpenAI, OpenAIEmbeddings

from langchain\_chroma import Chroma

from langchain.schema import Document

from langchain\_core.prompts import ChatPromptTemplate, MessagesPlaceholder

from langchain\_core.output\_parsers import StrOutputParser

from langchain\_core.runnables import (

RunnableLambda, RunnableParallel, RunnableSequence, RunnablePassthrough

)

from langchain\_community.retrievers import BM25Retriever

from langchain.retrievers import EnsembleRetriever, ContextualCompressionRetriever

from langchain.retrievers.document\_compressors import LLMChainExtractor

# Optional: Self-Query retriever (metadata-aware NL → filters). Toggle in sidebar.

from langchain.retrievers.self\_query.base import SelfQueryRetriever

from langchain.retrievers.self\_query.schema import AttributeInfo

load\_dotenv()

# ----------------------- Config -----------------------

PERSIST\_DIR = "store/chroma"

COLL = "devops\_kb"

BM25\_CORPUS\_PATH = Path("store") / "bm25\_corpus.ndjson"

FEEDBACK\_LOG = Path("store") / "feedback.jsonl"

st.set\_page\_config(page\_title="DevOps Copilot (RAG)", page\_icon="🧩", layout="wide")

st.title("🧩 DevOps Copilot — RAG for SRE/Cloud Teams")

st.caption("Ask about runbooks, Terraform modules, SLOs, incidents, and release notes. Answers cite sources.")

# ----------------------- Auth (demo-level) -----------------------

token\_expected = os.getenv("APP\_AUTH\_TOKEN")

if token\_expected:

with st.sidebar:

st.subheader("Auth")

token\_entered = st.text\_input("Access token", type="password")

if token\_entered != token\_expected:

st.stop()

# ----------------------- Sidebar Settings -----------------------

with st.sidebar:

st.header("Settings")

model\_name = st.selectbox("LLM", ["gpt-4o-mini", "gpt-4o"], index=0)

temperature = st.slider("Temperature", 0.0, 1.0, 0.0, 0.1)

top\_k = st.slider("Top K (vector)", 1, 10, 4)

hybrid\_weight = st.slider("Hybrid weight (semantic vs keyword)", 0.0, 1.0, 0.65, 0.05)

use\_compression = st.checkbox("Contextual compression (LLM trims)", value=True)

use\_self\_query = st.checkbox("Metadata self-query (LLM builds filters)", value=False)

st.markdown("---")

st.markdown("\*\*Filters\*\* (applied if self-query off):")

team = st.selectbox("team", ["any","sre","platform","sec","data"], index=0)

service = st.text\_input("service contains", "")

env = st.selectbox("env", ["any","dev","staging","prod"], index=0)

st.markdown("---")

st.caption("Tip: Toggle compression if context is noisy or too long.")

# ----------------------- Cache resources -----------------------

@st.cache\_resource(show\_spinner=False)

def get\_llm(name: str, temp: float):

return ChatOpenAI(model=name, temperature=temp, streaming=False)

@st.cache\_resource(show\_spinner=False)

def get\_vectordb():

embeddings = OpenAIEmbeddings(model="text-embedding-3-small")

return Chroma(collection\_name=COLL, persist\_directory=PERSIST\_DIR, embedding\_function=embeddings)

@st.cache\_resource(show\_spinner=False)

def get\_bm25():

# Rehydrate BM25 corpus from ndjson

import json

if not BM25\_CORPUS\_PATH.exists():

raise RuntimeError("BM25 corpus missing. Run ingest.py first.")

docs: List[Document] = []

with BM25\_CORPUS\_PATH.open("r", encoding="utf-8") as f:

for line in f:

row = json.loads(line)

docs.append(Document(page\_content=row["text"], metadata=row.get("metadata", {})))

r = BM25Retriever.from\_documents(docs)

r.k = 6

return r

llm = get\_llm(model\_name, temperature)

vectordb = get\_vectordb()

bm25 = get\_bm25()

# ----------------------- Build retriever -----------------------

# Base vector retriever

vector\_retriever = vectordb.as\_retriever(

search\_type="similarity",

search\_kwargs={"k": top\_k}

)

# Optional basic metadata filter (when self-query is OFF)

def metadata\_filter():

filt: Dict[str, Any] = {}

if team != "any": filt["team"] = team

if env != "any": filt["env"] = env

if service:

# Chroma supports "where\_document" (text contains) or "where" (metadata exact)

# We'll use simple metadata startswith-ish by indexing service in metadata in ingestion.

filt["service"] = service

return filt

# Hybrid fusion via EnsembleRetriever

def build\_ensemble():

base = vector\_retriever

if not use\_self\_query and (team != "any" or env != "any" or service):

base = vectordb.as\_retriever(search\_kwargs={"k": top\_k, "filter": metadata\_filter()})

return EnsembleRetriever(retrievers=[base, bm25], weights=[hybrid\_weight, 1.0 - hybrid\_weight])

retriever = build\_ensemble()

# Optional contextual compression (LLM trims to most relevant parts)

if use\_compression:

compressor = LLMChainExtractor.from\_llm(llm)

retriever = ContextualCompressionRetriever(base\_retriever=retriever, base\_compressor=compressor)

# Optional Self-Query retriever (LLM interprets NL filters → metadata filters)

if use\_self\_query:

# Describe metadata schema for the LLM

attributes = [

AttributeInfo(name="team", description="Owning team such as sre, platform, sec, data", type="string"),

AttributeInfo(name="service", description="Service or component (e.g., redis, ingress, eks)", type="string"),

AttributeInfo(name="env", description="Deployment environment (dev, staging, prod)", type="string"),

AttributeInfo(name="confidentiality", description="Doc classification: public, internal, restricted", type="string"),

AttributeInfo(name="updated\_at", description="Last updated date in YYYY-MM-DD", type="string"),

]

retriever = SelfQueryRetriever.from\_llm(

llm=llm,

vectorstore=vectordb,

document\_contents="DevOps/SRE internal knowledge base",

metadata\_field\_info=attributes,

enable\_limit=True

)

# ----------------------- Prompt -----------------------

SYSTEM = (

"You are DevOps Copilot for SRE/platform engineers. "

"Use only the provided context to answer. If unsure, say you don't know. "

"Cite sources like [S1], [S2] aligned to the context items."

)

PROMPT = ChatPromptTemplate.from\_messages(

[

("system", SYSTEM),

MessagesPlaceholder("chat\_history"),

("human",

"Question: {question}\n\n"

"Context:\n{context}\n\n"

"Answer concisely (<= 6 sentences) with citations like [S1], [S2].")

]

)

# ----------------------- Helpers -----------------------

def docs\_to\_context(docs: List[Document]) -> str:

blocks = []

for i, d in enumerate(docs, 1):

meta = d.metadata or {}

src = meta.get("source", "unknown")

bits = [f"[S{i}] ({src})"]

tag = f"[team={meta.get('team','?')} service={meta.get('service','?')} env={meta.get('env','?')}]"

snippet = d.page\_content.strip().replace("\n", " ")

blocks.append(f"{bits[0]} {tag}\n{snippet}")

return "\n\n".join(blocks)

def cite\_list(docs: List[Document]) -> str:

lines = []

for i, d in enumerate(docs, 1):

meta = d.metadata or {}

src = meta.get("source", "unknown")

lines.append(f"[S{i}] {src} | team={meta.get('team','?')} service={meta.get('service','?')} env={meta.get('env','?')}")

return "\n".join(lines) if lines else "No sources."

def save\_feedback(question: str, answer: str, docs: List[Document], rating: str, note: str):

FEEDBACK\_LOG.parent.mkdir(parents=True, exist\_ok=True)

payload = {

"ts": time.time(),

"question": question,

"answer": answer,

"rating": rating,

"note": note,

"sources": [d.metadata for d in docs]

}

with FEEDBACK\_LOG.open("a", encoding="utf-8") as f:

f.write(json.dumps(payload, ensure\_ascii=False) + "\n")

# ----------------------- Chain (Runnable graph) -----------------------

def make\_chain():

# retrieve → context format

retrieve = RunnableLambda(lambda q: retriever.get\_relevant\_documents(q))

pack = RunnableLambda(lambda docs: {"docs": docs, "context": docs\_to\_context(docs)})

generator = RunnableSequence(

PROMPT,

llm,

StrOutputParser()

)

# fan-in answer + docs so UI can render both

final = RunnableParallel({"answer": generator, "docs": RunnablePassthrough()})

# build end-to-end pipeline

chain = RunnableSequence(

retrieve,

pack,

# carry forward original user input for the prompt

RunnableLambda(lambda d, \*\*kwargs: {"question": kwargs["\_\_input\_\_"], "context": d["context"], "docs": d["docs"]}),

final

)

return chain

chain = make\_chain()

# ----------------------- Chat state -----------------------

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

st.session\_state.history = [] # [{role, content}]

for msg in st.session\_state.history:

st.chat\_message("user" if msg["role"] == "user" else "assistant").markdown(msg["content"])

# ----------------------- Chat input -----------------------

user\_q = st.chat\_input("Ask about incidents, runbooks, Terraform, SLOs, etc.")

if user\_q:

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

st.chat\_message("user").markdown(user\_q)

# lightweight history string for the prompt

history\_text = "\n".join([f"{m['role'].upper()}: {m['content']}" for m in st.session\_state.history[-6:]])

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

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

result = chain.invoke({"question": user\_q, "chat\_history": history\_text})

answer = result["answer"]

docs = result["docs"]

st.markdown(answer)

with st.expander("Sources"):

st.code(cite\_list(docs))

# Feedback UI

st.markdown("\*\*Was this helpful?\*\*")

cols = st.columns([1,1,3])

with cols[0]:

if st.button("👍"):

save\_feedback(user\_q, answer, docs, "up", "")

st.success("Thanks!")

with cols[1]:

if st.button("👎"):

note = st.text\_input("What went wrong? (optional)", key=f"note-{len(st.session\_state.history)}")

save\_feedback(user\_q, answer, docs, "down", note)

st.info("Feedback recorded.")

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

**🐳 Dockerfile (prod-friendly)**

# Dockerfile

FROM python:3.11-slim

WORKDIR /app

ENV PYTHONDONTWRITEBYTECODE=1 \

PYTHONUNBUFFERED=1

RUN apt-get update && apt-get install -y build-essential && rm -rf /var/lib/apt/lists/\*

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

# Provide a non-root user (optional hardening)

RUN useradd -m appuser

USER appuser

# Default: run the app

EXPOSE 8501

CMD ["streamlit", "run", "app.py", "--server.address=0.0.0.0", "--browser.gatherUsageStats=false"]

**▶️ Run It**

# 1) install deps

pip install -r requirements.txt

# 2) add docs into ./data

# (markdown with optional YAML front matter for metadata is ideal)

# 3) ingest

python ingest.py

# 4) run app

# ensure you copied .env.example -> .env and set OPENAI\_API\_KEY

streamlit run app.py

**Docker:**

docker build -t devops-copilot .

docker run -p 8501:8501 --env-file .env -v $(pwd)/data:/app/data -v $(pwd)/store:/app/store devops-copilot

**🧩 What You’re Getting (Prod-grade RAG Features)**

* **Hybrid Retrieval**: Ensemble of vector similarity and BM25 keyword → better recall (handles acronyms like “EKS”, “SLO”).
* **Contextual Compression**: LLM shrinks noisy context to lower token cost and better focus.
* **Metadata Filtering**: Team/service/env/confidentiality via front-matter or path conventions.
* **Self-Query (optional)**: Users can type *“prod redis runbook from SRE updated after 2025-08-01”* and the LLM converts that into metadata filters.
* **Citations**: Every answer shows [S1], [S2] with file paths and tags.
* **Feedback Loop**: Simple 👍/👎 saved to store/feedback.jsonl for continuous improvement.
* **Persisted Vector DB (Chroma)**: restart-safe.
* **Containerized**: Ready for k8s/ecs.

**🛠️ Next Steps / Extensions**

* **Auth**: Replace demo token with SSO (Okta/Auth0) on a reverse proxy (Traefik/Nginx).
* **Observability**: Add LangChain tracing (LCEL callbacks) + Prometheus + logs.
* **Eval**: Nightly regression using curated Q/A pairs + RAGAS.
* **Re-rankers**: Add a cross-encoder reranker (e.g., Cohere Rerank) after base retrieval.
* **Tenancy**: Add org\_id in metadata and enforce filters server-side.
* **Editing**: Build an admin UI to re-ingest, delete, and tag documents.