# Build a Private Document-only Chatbot (Python + Streamlit)

I’ve put a complete step-by-step guide, runnable code snippets, and tips into this document. It shows how to ingest PDFs/DOCX, build a local embedding index (FAISS), search the index, and return relevant answers — all **offline** and **local** (no internet access required at runtime).

Below is the structure of the document you can find here:

## 1) Overview

* Goal: accept user questions in a Streamlit UI, search *only* inside user documents (PDF & DOCX), and return relevant answers. No calls to external web services.
* Main components: ingestion → chunking → embeddings (sentence-transformers) → vector store (FAISS) → retrieval → answer display or local LLM generation.

## 2) Architecture (short)

User (Streamlit) ---> Upload Docs ---> Ingest & Chunk --> Compute Embeddings --> FAISS Index  
 ^ |  
 | v  
 Query (text) ----> Embed Query --> Nearest Neighbors --> Return top-k passages  
 --> (Optionally) Local LLM generate answer

## 3) Key choices and offline considerations

* Use sentence-transformers for local embeddings (e.g. all-MiniLM-L6-v2) — downloads required once.
* Use faiss-cpu for vector search (entirely local index).
* For text extraction: PyPDF2 or pdfplumber for PDFs, and python-docx for DOCX.
* For generation (optional): transformers with a local model (small-medium) OR just return the top passages (safer & fast).
* All files, indexes, and models live on disk; privacy preserved.

## 4) Requirements (example requirements.txt)

streamlit  
sentence-transformers  
faiss-cpu  
PyPDF2  
python-docx  
transformers # optional, only if you want local generation  
torch # needed for transformers/sentence-transformers  
regex  
tqdm  
nltk # optional, for smarter sentence splitting  
pdfplumber # optional (better PDF text extraction than PyPDF2)

## 5) Ingesting documents (code)

* ingest.py — walks a folder or accepts uploaded files, extracts text, chunks with overlap, and stores chunks with metadata.

# ingest.py  
from pathlib import Path  
from typing import List, Dict  
import pdfplumber  
import docx  
  
def extract\_text\_from\_pdf(path: Path) -> str:  
 texts = []  
 with pdfplumber.open(path) as pdf:  
 for p in pdf.pages:  
 texts.append(p.extract\_text() or "")  
 return "\n".join(texts)  
  
def extract\_text\_from\_docx(path: Path) -> str:  
 doc = docx.Document(path)  
 return "\n".join(p.text for p in doc.paragraphs)  
  
# Simple chunker: chunk by characters with overlap  
def chunk\_text(text: str, chunk\_size:int=800, overlap:int=200) -> List[Dict]:  
 chunks = []  
 start = 0  
 n = len(text)  
 while start < n:  
 end = min(start + chunk\_size, n)  
 chunk = text[start:end].strip()  
 if chunk:  
 chunks.append({"text": chunk, "start": start, "end": end})  
 start += chunk\_size - overlap  
 return chunks

Notes: you can replace the char-based chunker with sentence-based chunking using nltk if you prefer.

## 6) Building embeddings and FAISS index

* index\_builder.py — create embeddings for each chunk and add to a FAISS index. Save index and a metadata mapping (list of dicts) to disk.

# index\_builder.py  
from sentence\_transformers import SentenceTransformer  
import numpy as np  
import faiss  
import pickle  
  
MODEL\_NAME = "all-MiniLM-L6-v2" # small, fast, good quality  
  
def build\_index(chunks, model\_name=MODEL\_NAME, index\_path="faiss\_index.bin", meta\_path="meta.pkl"):  
 model = SentenceTransformer(model\_name)  
 texts = [c['text'] for c in chunks]  
 embeddings = model.encode(texts, show\_progress\_bar=True, convert\_to\_numpy=True)  
  
 dim = embeddings.shape[1]  
 index = faiss.IndexFlatL2(dim)  
 index.add(embeddings)  
  
 faiss.write\_index(index, index\_path)  
 with open(meta\_path, 'wb') as f:  
 pickle.dump(chunks, f)  
  
 print(f"Saved FAISS index to {index\_path} and metadata to {meta\_path}")

Important: model.encode(..., convert\_to\_numpy=True) returns float32 vectors ready for FAISS.

## 7) Querying the index (search)

* search.py — embed the user query, search FAISS, return top-k results along with metadata.

# search.py  
from sentence\_transformers import SentenceTransformer  
import faiss  
import numpy as np  
import pickle  
  
MODEL\_NAME = "all-MiniLM-L6-v2"  
  
class Retriever:  
 def \_\_init\_\_(self, index\_path='faiss\_index.bin', meta\_path='meta.pkl', model\_name=MODEL\_NAME):  
 self.model = SentenceTransformer(model\_name)  
 self.index = faiss.read\_index(index\_path)  
 with open(meta\_path, 'rb') as f:  
 self.meta = pickle.load(f)  
  
 def retrieve(self, query:str, top\_k:int=5):  
 q\_emb = self.model.encode([query], convert\_to\_numpy=True)  
 D, I = self.index.search(q\_emb, top\_k)  
 results = []  
 for idx, dist in zip(I[0], D[0]):  
 meta = self.meta[idx]  
 results.append({"score": float(dist), "text": meta['text'], "meta": meta})  
 return results

## 8) Simple answering (no LLM): return top passages

* Safe, fast, and fully offline. Just display the nearest passages and let the user read them.

## 9) (Optional) Local LLM generation

* If you want a human-like answer, you can run a local LLM via transformers (CPU/GPU). Use a small model (e.g., distilgpt2 or another gpt2-family) if you only have CPU, or use local-llama variants if you have proper setup.
* Example: concatenate top-k passages into a context prompt and ask the local model to generate a concise answer. Beware: quality depends on model size.

# local\_gen.py  
from transformers import AutoTokenizer, AutoModelForCausalLM  
import torch  
  
MODEL = "distilgpt2" # example — small and quick  
  
def generate\_answer(context:str, question:str, model\_name=MODEL, max\_new\_tokens=150):  
 tokenizer = AutoTokenizer.from\_pretrained(model\_name)  
 model = AutoModelForCausalLM.from\_pretrained(model\_name)  
 prompt = f"Context:\n{context}\n\nQuestion: {question}\nAnswer:"  
 inputs = tokenizer(prompt, return\_tensors='pt')  
 outputs = model.generate(\*\*inputs, max\_new\_tokens=max\_new\_tokens, do\_sample=False)  
 return tokenizer.decode(outputs[0], skip\_special\_tokens=True)[len(prompt):].strip()

Note: from\_pretrained downloads model the first time — you’ll need internet to download models once. After that models are local.

## 10) Putting it together: streamlit\_app.py

* A Streamlit UI that accepts file uploads, indexes them (or loads existing index), and runs queries.

# streamlit\_app.py  
import streamlit as st  
from ingest import extract\_text\_from\_pdf, extract\_text\_from\_docx, chunk\_text  
from index\_builder import build\_index  
from search import Retriever  
import tempfile, os, pickle  
  
INDEX\_PATH = 'faiss\_index.bin'  
META\_PATH = 'meta.pkl'  
  
st.title('Private Document Chatbot (Local)')  
  
# Upload section  
uploaded\_files = st.file\_uploader('Upload PDF or DOCX', accept\_multiple\_files=True)  
if uploaded\_files:  
 all\_chunks = []  
 for uf in uploaded\_files:  
 suffix = uf.name.split('.')[-1].lower()  
 with tempfile.NamedTemporaryFile(delete=False, suffix='.'+suffix) as tmp:  
 tmp.write(uf.getbuffer())  
 tmp\_path = tmp.name  
 if suffix in ['pdf']:  
 text = extract\_text\_from\_pdf(tmp\_path)  
 elif suffix in ['docx']:  
 text = extract\_text\_from\_docx(tmp\_path)  
 else:  
 st.warning(f"Unsupported file type: {suffix}")  
 continue  
 chunks = chunk\_text(text)  
 # attach filename for traceability  
 for c in chunks:  
 c['source'] = uf.name  
 all\_chunks.extend(chunks)  
  
 if all\_chunks:  
 st.info('Building index — this may take a moment for many documents')  
 build\_index(all\_chunks, index\_path=INDEX\_PATH, meta\_path=META\_PATH)  
 st.success('Index built and saved locally.')  
  
# If index exists, load retriever  
if os.path.exists(INDEX\_PATH) and os.path.exists(META\_PATH):  
 retriever = Retriever(INDEX\_PATH, META\_PATH)  
 query = st.text\_input('Ask a question about your documents:')  
 top\_k = st.slider('Top K', min\_value=1, max\_value=10, value=5)  
  
 if st.button('Search') and query:  
 results = retriever.retrieve(query, top\_k=top\_k)  
 st.write('Top passages:')  
 for i, r in enumerate(results, 1):  
 st.markdown(f"\*\*Result {i} — score {r['score']:.4f} — source: {r['meta'].get('source','unknown')}\*\*")  
 st.write(r['text'])  
  
 # Optionally show generation toggle  
 if st.checkbox('Generate concise answer (uses a local LLM)'):  
 context = "\n\n".join([r['text'] for r in results])  
 with st.spinner('Generating...'):  
 from local\_gen import generate\_answer  
 answer = generate\_answer(context, query)  
 st.subheader('Generated Answer')  
 st.write(answer)

## 11) Saving and reusing indexes

* Save faiss\_index.bin and meta.pkl and re-use them across app restarts to avoid re-indexing every time.

## 12) Testing and evaluation

* Prepare a few test questions you know the answers to inside your docs. Check if top-k passages include the correct text.
* If answers are poor, try:
  + Smaller chunk\_size (more granular chunks)
  + Larger top\_k
  + Different embedding model

## 13) Performance tips

* Use faiss.IndexIVFFlat for larger datasets and faster search (needs training step). For small- to medium-sized collections, IndexFlatL2 is fine.
* Use batch encoding for faster embeddings.
* If memory is constrained, store embeddings on disk and load as needed.

## 14) Security & privacy

* Keep model files, indexes and documents on the machine. Do not call remote APIs.
* If you must share the app, run it on an internal network or add authentication.

## 15) Extras & improvements

* Add a UI to view indexed files and their chunk counts.
* Add fuzzy metadata search (file names, headings).
* Add semantic highlighting of returned passages.
* Add background worker to index large uploads (e.g. Celery or simple threading) — but ensure it runs locally.

## 16) Troubleshooting (common)

* PDFs with scanned images: need OCR (pytesseract) — this requires installing Tesseract locally.
* Very large PDFs: increase chunk\_size or process per-page.
* If sentence-transformers fails to download model: ensure initial run has internet to download weights, or pre-download model and place locally.

If you want, I can: - create the exact .py files (ingest.py, index\_builder.py, search.py, local\_gen.py, streamlit\_app.py, requirements.txt) and place them here in separate files; or - adapt the code to use pdfplumber vs PyPDF2; or - provide an option that *only* returns passages (no LLM) vs *with* local LLM (showing how to use transformers).

Let me know which of the above you’d like me to produce next and I will add the ready-to-run files.