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
```

```python
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.

```python
# 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.