**Project idea (suggested title):**  
**PaperSafe: Smart Document QA & Integrity Assistant** — an app that lets users upload research papers (one or many), ask fact-based questions, extract supporting sentences, and run light checks for potential plagiarism / duplicate content and dataset/metric extraction. Useful for conference organizers, reviewers, and researchers.

I’ll include:

* A full Streamlit app (app\_streamlit\_sih.py) — multi-PDF QA + structured JSON answers + supporting quote + simple plagiarism check (fuzzy overlap with indexed chunks + optional web-search disabled) + exportable audit logs.
* requirements.txt — minimal required packages.
* README.md — project overview, deployment instructions, SIH pitch bullets.
* Dockerfile — optional for deployment.
* Notes about secrets and scaling.

You can copy these files directly into your repo, commit & push, and deploy on Streamlit Cloud / Hugging Face Spaces or any server.

**1) app\_streamlit\_sih.py (main app)**

Save as app\_streamlit\_sih.py.

# app\_streamlit\_sih.py

"""

PaperSafe — Smart Document QA & Integrity Assistant (SIH 2025 demo)

Features:

- Multi-PDF upload + per-file indexing (FAISS + SentenceTransformers)

- Cross-document or per-document retrieval QA using Groq-hosted LLaMA

- Structured JSON answers with "answer", "supporting\_snippet", "source"

- Simple plagiarism / duplicate detection via fuzzy overlap of chunks

- Session Q/A history and CSV download

- Sidebar controls: chunk size, overlap, k, tokens, temperature

Notes:

- Put GROQ\_API\_KEY and GROQ\_API\_URL in Streamlit Secrets (or .env for local dev)

- Keep temperature low (0.0) for deterministic extraction

"""

from dotenv import load\_dotenv

load\_dotenv() # local dev only

import streamlit as st

import os, json, requests, hashlib, time

import pandas as pd

from typing import List, Dict

from langchain\_community.document\_loaders import PyPDFLoader

from langchain.text\_splitter import RecursiveCharacterTextSplitter

from langchain\_community.embeddings import SentenceTransformerEmbeddings

from langchain\_community.vectorstores import FAISS

from langchain.prompts import PromptTemplate

from difflib import SequenceMatcher

# ------------------ Config / Util ------------------

st.set\_page\_config(page\_title="PaperSafe (SIH2025)", layout="wide")

APP\_TITLE = "📚 PaperSafe — Document QA & Integrity Assistant (SIH 2025)"

st.title(APP\_TITLE)

# Helpers

def file\_hash(b: bytes) -> str:

return hashlib.sha256(b).hexdigest()[:16]

def similar(a: str, b: str) -> float:

return SequenceMatcher(None, a, b).ratio()

def parse\_json\_safe(raw: str):

try:

return json.loads(raw)

except Exception:

# attempt to find a JSON substring

s = raw.find("{")

e = raw.rfind("}")

if s != -1 and e != -1 and e > s:

try:

return json.loads(raw[s:e+1])

except Exception:

return None

return None

# ------------------ LLM wrapper (Groq) ------------------

class GroqRemoteLLM:

def \_\_init\_\_(self, api\_url=None, api\_key=None, model=None, timeout=60, temperature=0.0):

self.api\_url = api\_url or os.getenv("GROQ\_API\_URL")

self.api\_key = api\_key or os.getenv("GROQ\_API\_KEY")

self.model = model or os.getenv("GROQ\_MODEL", "llama-3.3-70b-versatile")

self.timeout = timeout

self.temperature = float(temperature)

def \_\_call\_\_(self, prompt: str, max\_tokens: int = 256) -> str:

if not self.api\_url or not self.api\_key:

raise RuntimeError("GROQ\_API\_URL or GROQ\_API\_KEY missing (set in env or Streamlit Secrets).")

headers = {"Authorization": f"Bearer {self.api\_key}", "Content-Type": "application/json"}

payload = {"model": self.model, "messages":[{"role":"system","content":"You are an information extraction assistant. Answer only from the context."},{"role":"user","content":prompt}], "max\_tokens": max\_tokens, "temperature": float(self.temperature)}

resp = requests.post(self.api\_url, json=payload, headers=headers, timeout=self.timeout)

resp.raise\_for\_status()

out = resp.json()

# robust extraction

if isinstance(out, dict):

choices = out.get("choices", [])

if choices and isinstance(choices, list):

first = choices[0]

if isinstance(first, dict):

msg = first.get("message")

if isinstance(msg, dict) and "content" in msg:

return msg["content"]

if "text" in first:

return first["text"]

return json.dumps(out)

# ------------------ Caching helpers ------------------

@st.cache\_resource

def get\_embeddings(model\_name="all-MiniLM-L6-v2"):

return SentenceTransformerEmbeddings(model\_name=model\_name)

@st.cache\_resource

def build\_faiss(\_docs, embed\_model\_name="all-MiniLM-L6-v2"):

emb = get\_embeddings(embed\_model\_name)

return FAISS.from\_documents(\_docs, emb)

# ------------------ Sidebar controls ------------------

st.sidebar.header("Settings")

CHUNK\_SIZE = st.sidebar.number\_input("Chunk size (chars)", min\_value=200, max\_value=2000, value=900, step=100)

CHUNK\_OVERLAP = st.sidebar.number\_input("Chunk overlap (chars)", min\_value=0, max\_value=400, value=80, step=10)

RETRIEVER\_K = st.sidebar.number\_input("Retriever k (per doc)", min\_value=1, max\_value=10, value=3, step=1)

MAX\_TOKENS = st.sidebar.number\_input("Max tokens (LLM)", min\_value=64, max\_value=1024, value=256, step=32)

TEMPERATURE = st.sidebar.slider("Temperature (determinism)", min\_value=0.0, max\_value=1.0, value=0.0, step=0.05)

EMBED\_MODEL = st.sidebar.selectbox("Embedding model", options=["all-MiniLM-L6-v2"], index=0)

st.sidebar.markdown("---")

st.sidebar.write("NOTE: Add GROQ\_API\_KEY and GROQ\_API\_URL in Streamlit Secrets (Manage app → Secrets).")

# ------------------ State init ------------------

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

st.session\_state["index"] = {} # cache\_key -> {filename, path, docs, faiss}

if "qa\_log" not in st.session\_state:

st.session\_state["qa\_log"] = []

# ------------------ UI: uploads ------------------

uploaded = st.file\_uploader("Upload one or more PDFs", type="pdf", accept\_multiple\_files=True)

if uploaded:

for f in uploaded:

try:

b = f.read()

fh = file\_hash(b)

cache\_key = f"{fh}-{CHUNK\_SIZE}-{CHUNK\_OVERLAP}-{EMBED\_MODEL}"

if cache\_key in st.session\_state["index"]:

st.info(f"{f.name} already indexed for current settings.")

continue

# save to disk

path = f"uploaded\_{fh}.pdf"

with open(path, "wb") as out:

out.write(b)

st.info(f"Indexing {f.name} ... (this may take time)")

loader = PyPDFLoader(path)

pages = loader.load()

splitter = RecursiveCharacterTextSplitter(chunk\_size=CHUNK\_SIZE, chunk\_overlap=CHUNK\_OVERLAP)

docs = splitter.split\_documents(pages)

faiss = build\_faiss(docs, embed\_model\_name=EMBED\_MODEL)

st.session\_state["index"][cache\_key] = {"filename": f.name, "path": path, "docs": docs, "faiss": faiss}

st.success(f"Indexed {f.name} — {len(docs)} chunks.")

except Exception as e:

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

# ------------------ Document selection ------------------

docs\_list = list(st.session\_state["index"].items())

if not docs\_list:

st.info("No documents indexed yet. Upload PDFs to begin.")

# Build selection options

doc\_options = ["All documents"]

for ck, meta in docs\_list:

doc\_options.append(f"{meta['filename']} ({len(meta['docs'])} chunks) | key:{ck[:8]}")

selected = st.selectbox("Select document (or All documents):", options=doc\_options)

# ------------------ Querying ------------------

query = st.text\_input("Ask a question about indexed documents")

if st.button("Run Query") and query:

# build list of selected cache\_keys

if selected == "All documents":

keys = list(st.session\_state["index"].keys())

else:

# find matching key

parts = selected.split("| key:")

if len(parts) == 2:

key\_suffix = parts[1].strip()

keys = [k for k in st.session\_state["index"].keys() if k.startswith(key\_suffix)]

if not keys:

# fallback try simpler match

keys = [k for k, m in st.session\_state["index"].items() if selected.startswith(m["filename"])]

else:

keys = list(st.session\_state["index"].keys())

if not keys:

st.error("No matching documents selected or indexed.")

else:

# Gather top chunks from each doc

aggregated = []

for ck in keys:

meta = st.session\_state["index"][ck]

faiss = meta["faiss"]

retriever = faiss.as\_retriever(search\_kwargs={"k": RETRIEVER\_K})

docs\_ret = retriever.get\_relevant\_documents(query)

for i, d in enumerate(docs\_ret):

aggregated.append({"cache\_key": ck, "filename": meta["filename"], "chunk\_index": i+1, "doc": d, "text": d.page\_content})

# sort optionally by nothing (FAISS ordering kept). Limit to top N

TOP\_N = min(8, len(aggregated))

top\_chunks = aggregated[:TOP\_N]

# Combine into context with separators + chunk labels

context = "\n\n----\n\n".join([f"DOCUMENT: {c['filename']} | CHUNK {c['chunk\_index']}\n{c['text']}" for c in top\_chunks])

# Build JSON extraction prompt asking for supporting snippet & source

prompt\_template = """

You are an information extraction assistant. Use ONLY the context below (top retrieved chunks) to answer the question.

Context:

{context}

Question:

{question}

Task:

1) Provide a concise factual answer (1-2 sentences) to the question.

2) Provide the exact supporting sentence from the context (field: supporting\_snippet).

3) Provide the source filename and chunk label in field: source (e.g. "paper.pdf | CHUNK 2").

Return ONLY a JSON object with keys: answer, supporting\_snippet, source.

If the answer is not present in the context, set "answer" to "Not stated in the document" and other fields to null.

"""

QA\_PROMPT = PromptTemplate(template=prompt\_template, input\_variables=["context", "question"])

prompt = QA\_PROMPT.format(context=context, question=query)

# Call LLM (Groq)

llm = GroqRemoteLLM(temperature=TEMPERATURE)

try:

with st.spinner("Querying model..."):

raw = llm(prompt, max\_tokens=MAX\_TOKENS)

except Exception as e:

st.error(f"LLM call failed: {e}")

raw = None

parsed = parse\_json\_safe(raw) if raw else None

# fallback if parsing fails: ask simpler extraction

if not parsed and raw:

try:

fallback\_prompt = f"Context:\n{context}\n\nQuestion: {query}\n\nAnswer in two lines: first the answer, second the supporting sentence (prefixed 'Support:')."

raw2 = llm(fallback\_prompt, max\_tokens=MAX\_TOKENS)

if "Support:" in raw2:

a, s = raw2.split("Support:", 1)

parsed = {"answer": a.strip(), "supporting\_snippet": s.strip(), "source": None}

else:

parsed = {"answer": raw2.strip(), "supporting\_snippet": None, "source": None}

except Exception:

parsed = None

# Show results

st.subheader("Model output (structured)")

if parsed:

st.json(parsed)

st.markdown("\*\*Answer:\*\*")

st.write(parsed.get("answer"))

if parsed.get("supporting\_snippet"):

with st.expander("Supporting snippet"):

st.write(parsed.get("supporting\_snippet"))

if parsed.get("source"):

st.caption(f"Source: {parsed.get('source')}")

else:

st.error("Model did not return parseable JSON. Raw output shown below.")

st.code(raw or "(no output)")

# Simple plagiarism / duplicate detection: check if supporting\_snippet appears (or similar) in other documents

try:

snippet = (parsed.get("supporting\_snippet") or "").strip()

duplicates = []

if snippet:

for ck, meta in st.session\_state["index"].items():

# check all chunks for similarity

for d in meta["docs"]:

sim = similar(snippet, d.page\_content[:len(snippet)+200]) # quick heuristic

if sim > 0.85 and meta["filename"] not in [x["filename"] for x in duplicates]:

duplicates.append({"filename": meta["filename"], "similarity": round(sim, 3)})

if duplicates:

st.warning("Possible duplicate content found in other documents:")

st.table(pd.DataFrame(duplicates))

else:

st.success("No obvious duplicates found for the supporting snippet among indexed docs.")

except Exception as e:

st.error("Error while checking duplicates: " + str(e))

# Save QA audit log

st.session\_state["qa\_log"].append({

"timestamp": time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime()),

"query": query,

"answer": parsed.get("answer") if parsed else (raw or ""),

"supporting\_snippet": parsed.get("supporting\_snippet") if parsed else None,

"source": parsed.get("source") if parsed else None,

"top\_docs": ";".join({c["filename"] for c in top\_chunks})

})

# ------------------ History & export ------------------

st.markdown("---")

st.subheader("Session QA Audit Log")

if st.session\_state["qa\_log"]:

df = pd.DataFrame(st.session\_state["qa\_log"])

st.dataframe(df.sort\_values(by="timestamp", ascending=False))

b = df.to\_csv(index=False).encode("utf-8")

st.download\_button("Download audit log (CSV)", data=b, file\_name="papersafe\_audit\_log.csv", mime="text/csv")

else:

st.write("No QA events in this session yet.")

# Debug toggle

if st.sidebar.checkbox("Show debug info"):

st.sidebar.write("Indexed docs keys:", list(st.session\_state["index"].keys()))

st.sidebar.write("Env GROQ\_API\_URL set:", bool(os.getenv("GROQ\_API\_URL")))

st.sidebar.write("Number of indexed documents:", sum(len(meta["docs"]) for meta in st.session\_state["index"].values()) if st.session\_state["index"] else 0)

**2) requirements.txt**

Save as requirements.txt.

streamlit

langchain

langchain-community

sentence-transformers

faiss-cpu

pypdf

requests

python-dotenv

pandas

Note: sentence-transformers will pull torch. For small demo usage on CPU it's OK but indexing may be slow. If you want a smaller dependency footprint, use a server with GPU or use smaller embedding models (but the demo uses all-MiniLM-L6-v2).

**3) README.md**

Save as README.md.

# PaperSafe — Smart Document QA & Integrity Assistant (SIH 2025)

## Project summary (SIH 2025)

PaperSafe helps conference organizers, reviewers and researchers quickly query uploaded papers, obtain concise factual answers with supporting text, and perform a light integrity check to detect duplicate content across submitted documents.

### Key features

- Multi-PDF upload and per-document FAISS indexing (embeddings cached)

- Retrieval-based QA using Groq-hosted LLaMA (model: `llama-3.3-70b-versatile`)

- Structured JSON answers with supporting sentence and source

- Simple duplicate detection across indexed documents

- Exportable audit log for review workflows

### Tech stack

- Python + Streamlit (UI)

- LangChain + FAISS + SentenceTransformers (embeddings + retrieval)

- Groq inference API (LLaMA 3.3 70B) for extraction

- Deployable on Streamlit Cloud, HuggingFace Spaces, or Docker

### Getting started (local)

1. Clone repo and create an environment:

```bash

conda create -n papersafe python=3.10 -y

conda activate papersafe

pip install -r requirements.txt

Create .env with:

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

GROQ\_API\_URL=https://api.groq.com/openai/v1/chat/completions

GROQ\_MODEL=llama-3.3-70b-versatile

Run:

streamlit run app\_streamlit\_sih.py

**Deployment**

* For Streamlit Cloud / Hugging Face Spaces: push repo to GitHub and set app entrypoint to app\_streamlit\_sih.py. Store secrets in the platform's Secrets UI (do not commit .env).

**SIH Notes**

* This demo shows a reproducible workflow for document QA + light integrity checks. For production SIH builds, add:
  + Persistent FAISS storage (S3) to avoid re-indexing
  + User auth & role-based access
  + Scalable backend (FastAPI + background indexing)
  + Stronger plagiarism detection (external corpora, crossref, similarity thresholds)

**License**

MIT

---

## 4) `Dockerfile` (optional)

Save as `Dockerfile` to containerize:

```dockerfile

FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .

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

COPY . .

EXPOSE 8080

CMD ["streamlit", "run", "app\_streamlit\_sih.py", "--server.port=8080", "--server.address=0.0.0.0"]

**5) SIH Submission Pitch (short)**

Include this in your SIH docs:

* **Problem:** reviewers need tools to quickly triage and verify many paper submissions; manual review is slow and error-prone.
* **Solution:** PaperSafe enables quick factual extraction and integrity checks, producing audit logs and highlighting supporting evidence.
* **Impact:** speeds up review cycles, increases reviewer confidence, flags suspicious duplicates early.
* **Novelty:** Retrieval + LLM extraction with supporting snippet + cross-document duplication detection in one integrated tool.

**6) Next steps & optional enhancements (choose 1–3)**

* Add **persistent index storage** (S3 or GCS) so indexes survive restarts. I can provide code for saving/loading FAISS to S3.
* Replace SentenceTransformer embeddings with a **cloud embedding API** (if you want faster indexing).
* Add **user authentication** and job-queue for background indexing (Celery/RQ + Redis).
* Add **automated evaluation**: load gold Q/A CSV and compute EM / fuzzy scores across queries.

Tell me which enhancements you want next and I’ll produce the code (e.g., S3 persistence, FastAPI wrapper, or a CI test script).

**Security & Cost reminders**

* Don’t commit GROQ API keys to GitHub. Use Streamlit Secrets or environment variables. Rotate keys if accidentally leaked.
* Groq model usage may incur cost; keep max\_tokens low and temperature=0 for extraction tasks.

If you want, I can:

* Provide a **one-click GitHub repo template** (all files already populated) and exact git commands to push, or
* Generate a **short demo video script / slide deck** for SIH judges explaining the app.