I am working as a shadow product manager and building a portfolio of products for my interview, for the uploaded project tasks, i have completed till Day 5, now help break down DAY 6 in easy steps with easy explanation as well and codes

Day 6 focuses on creating a local vector database and embedding the de-identified note chunks so retrieval will work on Day 7. Here is a clear, step-by-step breakdown with simple explanations and ready-to-run Python scripts, aligned exactly with the Day 6 objectives and artifacts in the uploaded plan.[1]

What to do today

Create chunking + embedding + vector DB persistence so de-identified notes from Day 5 can be searched semantically, producing indexer.py and a local stored index.[1]

Folder setup

- Use existing folders from earlier days: /app, /data/synthetic_notes, /data/outputs, /secure_store.[1]
- Create a new folder for the vector DB, for example /data/vector_store (auto-created by the script).[1]

Step 1: Install required libraries

- Two solid choices: FAISS (fast) or Chroma (easy persistence) as per the plan; embeddings via sentence-transformers/all-MiniLM-L6-v2 or Bio_ClinicalBERT for clinical flavor.[1]
- Commands:
  - pip install faiss-cpu sentence-transformers chromadb tqdm pyyaml.[1]

Step 2: Choose the vector store

- Option A (recommended for simplicity): Chroma persists to disk automatically (good for demos).[1]
- Option B: FAISS for speed; persistence requires saving index files and metadata; both are supported below.[1]

Step 3: Prepare inputs

- Use de-identified text outputs from Day 5 as the only source for indexing to stay HIPAA-aligned; do not index original PHI.[1]
- Ensure /data/outputs contains de-identified .txt files or JSON with de-identified text fields; the script supports .txt and .json.[1]

Step 4: Chunking strategy

- Chunk to 200–400 tokens with a small overlap (e.g., 40–60) to preserve context across boundaries, matching the Day 6 guidance.[1]
- Simple approach: word-based chunking with an approximate token size; this is sufficient for the MVP.[1]

Step 5: Embedding model

- Default: sentence-transformers/all-MiniLM-L6-v2 (fast, 384-dim).[1]
- Optional: swap to Bio_ClinicalBERT embeddings if desired for domain semantics later (slower, extra setup).[1]

Step 6: Run indexer.py to build the DB

- The script:
  - Loads de-identified notes from /data/outputs.
  - Chunks each note into 200–400 token-like segments.
  - Computes embeddings.
  - Writes to Chroma (disk) or FAISS files with metadata like note_id and section.[1]

Step 7: Verify

- After running, confirm a persisted database exists (e.g., /data/vector_store/chroma.sqlite for Chroma or .faiss + .pkl files for FAISS).[1]
- Keep this for Day 7 retriever + RAG pipeline.[1]

indexer.py (drop-in, end-to-end)

- Place this in app/indexer.py and run: python app/indexer.py —input_dir ./data/outputs —db_type chroma —persist_dir ./data/vector_store.[1]

```
# app/indexer.py
# Day 6: Vector store & embeddings
# Usage examples:
#   python app/indexer.py --input_dir ./data/outputs --db_type chroma --persist_dir ./dat
#   python app/indexer.py --input_dir ./data/outputs --db_type faiss  --persist_dir ./dat

import os
import json
import argparse
from pathlib import Path
from typing import List, Dict, Tuple
from tqdm import tqdm

# Embeddings
from sentence_transformers import SentenceTransformer

# Vector stores
# Chroma
import chromadb
from chromadb.config import Settings as ChromaSettings

# FAISS
import faiss
import pickle

DEFAULT_CHUNK_TOKENS = 300
DEFAULT_OVERLAP_TOKENS = 50

def read_note_files(input_dir: str) -> List[Dict]:
    """
    Reads de-identified notes from .txt or .json in input_dir.
    Expects .json to have a 'text' field containing de-identified content.
    Returns list of dicts: {id, text, section?}
    """
    items = []
    p = Path(input_dir)
    if not p.exists():
        raise FileNotFoundError(f"Input dir not found: {input_dir}")

    for fp in p.glob("**/*"):
        if fp.is_dir():
            continue
        if fp.suffix.lower() == ".txt":
            text = fp.read_text(encoding="utf-8", errors="ignore").strip()
            if text:
                items.append({"id": fp.stem, "text": text, "section": None})
        elif fp.suffix.lower() == ".json":
            try:
                obj = json.loads(fp.read_text(encoding="utf-8", errors="ignore"))
                text = obj.get("text") or obj.get("deidentified_text") or ""
                section = obj.get("section")
                if text:
                    items.append({"id": fp.stem, "text": text.strip(), "section": sectior
            except Exception:
                # Skip malformed
                continue
    return items

def approx_tokenize(text: str) -> List[str]:
    """
    Approximate tokenization by splitting on whitespace.
    For MVP this is fine; can replace with tiktoken later.
    """
    return text.split()
```

```python
def detokenize(tokens: List[str]) -> str:
    return " ".join(tokens)

def chunk_text(text: str, chunk_tokens: int, overlap_tokens: int) -> List[str]:
    """
    Simple sliding window chunking.
    """
    tokens = approx_tokenize(text)
    chunks = []
    i = 0
    n = len(tokens)
    while i < n:
        j = min(i + chunk_tokens, n)
        chunk = detokenize(tokens[i:j])
        if chunk.strip():
            chunks.append(chunk)
        if j == n:
            break
        i = j - overlap_tokens
        if i < 0:
            i = 0
    return chunks

def embed_texts(model: SentenceTransformer, texts: List[str]):
    return model.encode(texts, show_progress_bar=False, convert_to_numpy=True, normalize_

def build_chroma(persist_dir: str, collection_name: str = "notes"):
    client = chromadb.PersistentClient(
        path=persist_dir,
        settings=ChromaSettings(allow_reset=True)
    )
    if collection_name in [c.name for c in client.list_collections()]:
        coll = client.get_collection(collection_name)
    else:
        coll = client.create_collection(collection_name)
    return client, coll

def save_faiss(index, vectors_meta: List[Dict], persist_dir: str):
    os.makedirs(persist_dir, exist_ok=True)
    faiss_path = os.path.join(persist_dir, "index.faiss")
    meta_path = os.path.join(persist_dir, "meta.pkl")
    faiss.write_index(index, faiss_path)
    with open(meta_path, "wb") as f:
        pickle.dump(vectors_meta, f)

def load_faiss(persist_dir: str):
    faiss_path = os.path.join(persist_dir, "index.faiss")
    meta_path = os.path.join(persist_dir, "meta.pkl")
    if os.path.exists(faiss_path) and os.path.exists(meta_path):
        index = faiss.read_index(faiss_path)
        with open(meta_path, "rb") as f:
            meta = pickle.load(f)
        return index, meta
    return None, []
```

```
def main():
    parser = argparse.ArgumentParser(description="Day 6: Build local vector DB from de-ic
    parser.add_argument("--input_dir", required=True, help="Directory with de-identified
    parser.add_argument("--persist_dir", default="./data/vector_store", help="Where to pe
    parser.add_argument("--db_type", choices=["chroma", "faiss"], default="chroma", help=
    parser.add_argument("--model_name", default="sentence-transformers/all-MiniLM-L6-v2",
    parser.add_argument("--chunk_tokens", type=int, default=DEFAULT_CHUNK_TOKENS, help="A
    parser.add_argument("--overlap_tokens", type=int, default=DEFAULT_OVERLAP_TOKENS, hel
    parser.add_argument("--collection", default="notes", help="Collection name (Chroma).'
    args = parser.parse_args()

    notes = read_note_files(args.input_dir)
    if not notes:
        print(f"No de-identified notes found in {args.input_dir}. Ensure Day 5 outputs ex
        return

    print(f"Loaded {len(notes)} de-identified notes from {args.input_dir}")
    os.makedirs(args.persist_dir, exist_ok=True)

    print(f"Loading embedding model: {args.model_name}")
    model = SentenceTransformer(args.model_name)

    all_chunk_texts = []
    all_chunk_ids = []
    all_metadata = []

    print("Chunking notes...")
    for note in tqdm(notes):
        chunks = chunk_text(note["text"], args.chunk_tokens, args.overlap_tokens)
        for idx, ch in enumerate(chunks):
            cid = f"{note['id']}::chunk_{idx}"
            all_chunk_texts.append(ch)
            all_chunk_ids.append(cid)
            all_metadata.append({
                "note_id": note["id"],
                "chunk_index": idx,
                "section": note.get("section")
            })

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

    print("Embedding chunks...")
    vectors = embed_texts(model, all_chunk_texts)

    if args.db_type == "chroma":
        print("Building Chroma persistent collection...")
        client, coll = build_chroma(args.persist_dir, args.collection)

        # Upsert in manageable batches
        batch = 512
        for i in tqdm(range(0, len(all_chunk_texts), batch)):
            j = min(i + batch, len(all_chunk_texts))
            coll.upsert(
                ids=all_chunk_ids[i:j],
                embeddings=vectors[i:j].tolist(),
                documents=all_chunk_texts[i:j],
                metadatas=all_metadata[i:j],
            )
        print(f"Chroma collection '{args.collection}' persisted at {args.persist_dir}")

    elif args.db_type == "faiss":
        print("Building FAISS index...")
        d = vectors.shape[^1_1]
        index = faiss.IndexFlatIP(d)  # normalized vectors → use inner product as cosine
        # Try to load existing
        existing_index, existing_meta = load_faiss(args.persist_dir)
        if existing_index is not None:
            print("Appending to existing FAISS index...")
            index = existing_index
            vectors_meta = existing_meta
        else:
            vectors_meta = []
        index.add(vectors)
        vectors_meta.extend([
            {
                "id": all_chunk_ids[k],
                "text": all_chunk_texts[k],
                "meta": all_metadata[k]
            } for k in range(len(all_chunk_texts))
        ])
        save_faiss(index, vectors_meta, args.persist_dir)
        print(f"FAISS index persisted at {args.persist_dir}")

    print("Done.")

if __name__ == "__main__":
    main()
```

How to run

- For Chroma (recommended):
    - python app/indexer.py —input_dir ./data/outputs —db_type chroma —persist_dir ./data/vector_store —collection notes.[1]
- For FAISS:
    - python app/indexer.py —input_dir ./data/outputs —db_type faiss —persist_dir ./data/vector_store_faiss.[1]

What this produces

- An artifact indexer.py and a persisted local vector DB containing chunk embeddings with metadata (note_id, section), exactly as required by Day 6.[1]

Tips and next steps

- Keep chunk_tokens between 200–400 and overlap 40–60 to match Day 6 guidance for retrieval quality tomorrow.[1]
- Tomorrow (Day 7), wire a retriever using this DB (Haystack or LangChain) and feed top_k results into a generator for structured summaries.[1]

⁂

can you now go to Day 7 and provide all details and explanation what we are trying to acheive and code sets as well to execute broken down by steps

Day 7 delivers the first working retriever and a minimal RAG pipeline that takes a query, pulls the most relevant de-identified chunks from the Day 6 vector store, and prepares them for summarization on Day 8.[2]

Goal for Day 7

Build a retriever over the existing vector DB (Chroma or FAISS) and expose a simple CLI that takes a prompt like "Summarize the note into HPI/Assessment/Plan" and returns top-k context snippets for the generator to use tomorrow.[2]

What gets built

- A minimal RAG pipeline file app/rag_pipeline.py with:
  - Loader for the persisted vector DB from Day 6.[2]
  - Retriever function to run semantic search over de-identified chunks only.[2]
  - CLI to accept a user question and print ranked snippets with their metadata, ready to be fed into a summarizer (Day 8).[2]

Prerequisites

- Day 6 index exists under ./data/vector_store (Chroma) or ./data/vector_store_faiss (FAISS) with chunk embeddings and metadata stored.[2]
- Installed libraries include haystack-ai or langchain, sentence-transformers, chromadb, faiss-cpu from earlier steps.[2]

Step-by-step plan

1) Pick a framework (Haystack or LangChain)

- Haystack path: define a DocumentStore (Chroma/FAISS), a DensePassageRetriever or EmbeddingRetriever aligned with Day 6 embeddings, and a simple pipeline for retrieval-only outputs today.[2]
- LangChain path: load the persisted vector store and use VectorStoreRetriever to return top-k Document objects with text + metadata.[2]

Assumption: Use LangChain for fewer moving parts in a small repo; Haystack snippet also provided for teams preferring pipelines.[2]

2) Retrieval choices to match Day 6

- If Day 6 used Chroma, load the same persist_directory and collection name; query with cosine similarity on normalized vectors.[2]
- If Day 6 used FAISS, load index + metadata (saved .faiss and .pkl) and wrap it with a simple retriever interface.[2]

3) Querying strategy

- Start with top_k=5 and a semantic query like “Generate HPI/Assessment/Plan from this patient’s note” or “Extract Assessment and Plan only” for testing.[2]
- Do not generate a summary yet; only return the stitched context to feed Day 8’s model with a structured prompt.[2]

4) Artifact to produce

- app/rag_pipeline.py with a CLI:
  - python app/rag_pipeline.py --db_type chroma --persist_dir ./data/vector_store --collection notes --query "Summarize into HPI/Assessment/Plan" --top_k 5.[2]

Option A: LangChain implementation (recommended)

```
# app/rag_pipeline.py
# Day 7: Retriever + RAG baseline (retrieval only; generation comes on Day 8)
# Example usage:
#   python app/rag_pipeline.py --db_type chroma --persist_dir ./data/vector_store --colle
#   python app/rag_pipeline.py --db_type faiss  --persist_dir ./data/vector_store_faiss -

import os
import argparse
import pickle
from typing import List, Dict

from sentence_transformers import SentenceTransformer
import numpy as np

# LangChain vector store wrappers
from langchain_community.vectorstores import Chroma, FAISS
from langchain_core.documents import Document
```

```
# For FAISS manual load if using custom persisted index
import faiss

def load_embedder(model_name: str = "sentence-transformers/all-MiniLM-L6-v2"):
    model = SentenceTransformer(model_name)
    def embed_f(texts: List[str]) -> List[List[float]]:
        vecs = model.encode(texts, convert_to_numpy=True, normalize_embeddings=True)
        return vecs.tolist()
    return model, embed_f

def load_chroma(persist_dir: str, collection: str, embed_f):
    # LangChain Chroma requires an embedding function wrapper
    from langchain.embeddings.base import Embeddings
    class STEmbeddings(Embeddings):
        def embed_documents(self, texts: List[str]) -> List[List[float]]:
            return embed_f(texts)
        def embed_query(self, text: str) -> List[float]:
            return embed_f([text])[^2_0]

    embeddings = STEmbeddings()
    vectordb = Chroma(
        collection_name=collection,
        persist_directory=persist_dir,
        embedding_function=embeddings
    )
    return vectordb

def load_faiss_langchain(persist_dir: str, embed_f):
    # If Day 6 saved FAISS with LangChain’s FAISS.save_local, we can do:
    # return FAISS.load_local(persist_dir, embeddings, allow_dangerous_deserialization=Tr
    # But Day 6 saved raw FAISS + meta.pkl; handle that manually and wrap.
    from langchain.embeddings.base import Embeddings
    class STEmbeddings(Embeddings):
        def embed_documents(self, texts: List[str]) -> List[List[float]]:
            return embed_f(texts)
        def embed_query(self, text: str) -> List[float]:
            return embed_f([text])[^2_0]
    embeddings = STEmbeddings()

    index_path = os.path.join(persist_dir, "index.faiss")
    meta_path = os.path.join(persist_dir, "meta.pkl")
    if not (os.path.exists(index_path) and os.path.exists(meta_path)):
        raise FileNotFoundError(f"FAISS files not found in {persist_dir}")

    index = faiss.read_index(index_path)
    with open(meta_path, "rb") as f:
        meta = pickle.load(f)

    # Build FAISS VectorStore from texts + metadata to leverage LC retriever
    texts = [m["text"] for m in meta]
    metadatas = [m["meta"] | {"id": m["id"]} for m in meta]
    vectordb = FAISS.from_texts(texts=texts, embedding=embeddings, metadatas=metadatas)
    # Replace the underlying index with prebuilt (saves re-embedding cost when querying)
    vectordb.index = index
    return vectordb
```

```
def retrieve(vdb, query: str, top_k: int = 5):
    retriever = vdb.as_retriever(search_kwargs={"k": top_k})
    docs: List[Document] = retriever.invoke(query)
    return docs

def format_context(docs: List[Document]) -> str:
    parts = []
    for i, d in enumerate(docs, 1):
        md = d.metadata or {}
        parts.append(f"[{i}] note_id={md.get('note_id')} section={md.get('section')} chur
    return "\n\n---\n\n".join(parts)

def main():
    parser = argparse.ArgumentParser(description="Day 7: Retriever + RAG baseline (retrie
    parser.add_argument("--db_type", choices=["chroma", "faiss"], default="chroma")
    parser.add_argument("--persist_dir", default="./data/vector_store")
    parser.add_argument("--collection", default="notes")
    parser.add_argument("--model_name", default="sentence-transformers/all-MiniLM-L6-v2")
    parser.add_argument("--query", required=True)
    parser.add_argument("--top_k", type=int, default=5)
    args = parser.parse_args()

    _, embed_f = load_embedder(args.model_name)

    if args.db_type == "chroma":
        vdb = load_chroma(args.persist_dir, args.collection, embed_f)
    else:
        vdb = load_faiss_langchain(args.persist_dir, embed_f)

    docs = retrieve(vdb, args.query, args.top_k)
    context = format_context(docs)
    print("\n=== Retrieved Context (to feed Day 8 summarizer) ===\n")
    print(context)

if __name__ == "__main__":
    main()
```

How to run:

- Chroma: python app/rag_pipeline.py --db_type chroma --persist_dir ./data/vector_store --collection notes --query "Summarize into HPI/Assessment/Plan" --top_k 5.[2]
- FAISS: python app/rag_pipeline.py --db_type faiss --persist_dir ./data/vector_store_faiss --query "Extract Assessment and Plan" --top_k 5.[2]

What this returns:

- A ranked set of top-k chunks with note_id, section, and chunk_index so the generator can respect structure and avoid hallucinating beyond the provided context tomorrow.[2]

Option B: Haystack implementation (alternative)

```
# app/rag_pipeline_haystack.py
# Retrieval-only pipeline using Haystack
# Usage:
#   python app/rag_pipeline_haystack.py --db_type chroma --persist_dir ./data/vector_stor

import os
import argparse
from typing import List

from sentence_transformers import SentenceTransformer
import numpy as np

from haystack.document_stores import FAISSDocumentStore
from haystack.document_stores.types import DuplicatePolicy
from haystack import Pipeline
from haystack.components.retrievers import EmbeddingRetriever
from haystack.dataclasses import Document

def load_embedder(model_name: str = "sentence-transformers/all-MiniLM-L6-v2"):
    model = SentenceTransformer(model_name)
    def embed_f(texts: List[str]):
        vecs = model.encode(texts, convert_to_numpy=True, normalize_embeddings=True)
        return vecs
    return model, embed_f

def load_faiss_document_store(persist_dir: str):
    # For a fresh Haystack store, we'd normally write docs; here we assume rebuilding fro
    # In portfolio context, rebuilding a Haystack store from saved chunks/metadata is acc
    return FAISSDocumentStore(embedding_dim=384, similarity="cosine")

def main():
    parser = argparse.ArgumentParser(description="Day 7: Retriever baseline with Haystack
    parser.add_argument("--persist_dir", default="./data/vector_store_faiss")
    parser.add_argument("--query", required=True)
    parser.add_argument("--top_k", type=int, default=5)
    parser.add_argument("--model_name", default="sentence-transformers/all-MiniLM-L6-v2")
    args = parser.parse_args()

    _, embed_f = load_embedder(args.model_name)
    # In a full integration, ingest Day 6 chunks into Haystack's store once, then query.
    # For brevity, assume a small demo where texts+meta are reloaded and written to the s

    # Minimal demo: create an empty store and show how to query after writing docs
    store = load_faiss_document_store(args.persist_dir)
    retriever = EmbeddingRetriever(document_store=store, embedding_model=args.model_name,

    # If you already ingested, skip writing. Otherwise, read meta.pkl and add_documents h
    meta_path = os.path.join(args.persist_dir, "meta.pkl")
    if os.path.exists(meta_path):
        import pickle
        with open(meta_path, "rb") as f:
            meta = pickle.load(f)
        docs = [Document(content=m["text"], meta=m["meta"] | {"id": m["id"]}) for m in me
        store.write_documents(docs, policy=DuplicatePolicy.OVERWRITE)
        store.update_embeddings(retriever)

    pipe = Pipeline()
    pipe.add_component("retriever", retriever)
    result = pipe.run(data={"retriever": {"query": args.query, "top_k": args.top_k}})
    hits = result["retriever"]["documents"]

    print("\n=== Retrieved Context (Haystack) ===\n")
    for i, d in enumerate(hits, 1):
        print(f"[{i}] note_id={d.meta.get('note_id')} section={d.meta.get('section')} chu
        print(d.content)
        print("\n---\n")

if __name__ == "__main__":
    main()
```

Validation checklist

- Query returns only de-identified content; no PHI comes from the index by design (enforced since Day 5/6).[2]
- Top-k is tunable; start with 5 and adjust based on recall for HPI/Assessment/Plan queries.[2]
- Metadata (note_id, section, chunk_index) is preserved in outputs to support structured prompts on Day 8.[2]

What Day 8 will add

- A summarization model (BART or FLAN-T5) and a strict prompt template enforcing clinical sections and “answer only from context,” consuming the retrieved snippets from today’s pipeline.[2]

⁂

1. COMPLETE-WEEK-BY-WEEK-TASKS.docx
2. COMPLETE-WEEK-BY-WEEK-TASKS.docx