Project Name: SwiftVisa AI-Based Visa Eligibility Screening Agent

Project summary (one line)

SwiftVisa is a retrieval-augmented screening agent that indexes official visa-policy documents per country into a vector database (FAISS) so the agent can quickly retrieve relevant policy chunks for eligibility-checking or question answering.

Milestone 1: Create a vector DB

- **Goal**

  Create a pipeline for pre-processing the documents related to visa eligibility related criteria, convert them as embeddings, and store them in a vector database.

- **List of steps to achieve the goal:**
  1. create a folder with the list of countries that agent caters
  2. Preprocess/ embedding of the documents using the sentence transformer
  3. Store this vector embedding in FAISS

Implementation Steps:

1. Create a Data/ folder that contains documents (PDF and TXT) grouped by country (e.g., Canada.pdf, US.pdf, UK.pdf, ...).
2. Preprocess text and split it into semantically useful chunks (chunking).
3. Generate dense **sentence embeddings** for each chunk using a sentence-transformer model.
4. Store embeddings and metadata in **FAISS**, using an ID map so each chunk has a globally unique unique_id.
5. Persist metadata and raw chunks so retrieval can show the originating document and the chunk text.

Deliverables produced (code files)

Files you currently have and used in the pipeline:

- *nltk_setup.py* — ensures NLTK punkt tokenizer is present.
- *pdf_utils.py* — functions to read PDFs and .txt files and clean text.

- *chunking.py* — sentence-aware chunking into at most max_tokens windows.
- *embedding.py* — loads sentence-transformers/all-MiniLM-L6-v2, and embed the chunks.
- *vector_store.py* — builds FAISS IndexFlatIP wrapped in IndexIDMap, saves index, and writes metadata JSON keyed by unique_id.
- *main.py* —

  orchestrator:

  finds files in Data/, performs extraction → cleaning → chunking → embedding → builds the FAISS index.
- *demonstration.py* — demo comparing FAISS search vs brute-force list search.

Learnings from week 1:

1. **Retrieval-Augmented Generation (RAG)**

- **What is RAG?**

  Retrieval-Augmented Generation (RAG) is an AI architecture where a Large Language Model (LLM) is combined with an external knowledge retrieval system.

  Instead of relying only on the LLM's internal knowledge, RAG retrieves **relevant documents**, **embeddings**, or **chunks** from a database and provides them as context to the model.

- **Why RAG is used:**

  As LLMs have limitations. They cannot memorize everything. Their knowledge is static. Also, they may hallucinate.

  RAG fixes this.

- **How RAG works?**

  1. **User asks a query**
     Example: "Give me the refund policy for Product X."
  2. **Embedding + Retrieval**
     - Convert user query → **embedding vector**
     - Perform internal multiplication in a vector database (FAISS/Chroma)

    - Retrieve most relevant chunks/documents

3. **Augment LLM Input**
   Send the retrieved text + original query to the LLM.

4. **LLM Generates a grounded response**
   So the output is *accurate and based on external verified information*.

2. What are embeddings?

- **Definition**

  Embeddings are numerical vector representations of data. They capture semantic meaning, context, and relationships.

  Example: "car" and "vehicle" have embedding vectors close to each other.

- **Methods to search for similarity between embeddings:**
  - Euclidean distance between query embedding vs. sentence/document embeddings.
  - Cosine similarity (angle) is common to measure semantic similarity.
  - Inner product (dot product) equals cosine if vectors are L2-normalized.

3. Different embedding generating models.

- **OpenAI Embeddings**
  Examples:
  - text-embedding-ada-002
  - text-embedding-3-large
  - text-embedding-3-small

- **Google Embedding Models (Gemini)**

- **BERT-based Embedding Models**
  These are Transformer-based:
  - BERT
  - RoBERTa
  - DistilBERT
  - ALBERT

- **Sentence-BERT (SBERT)**
  (Specially optimized to produce semantic embeddings)
  Models include:
  - all-MiniLM-L6-v2

  - all-mpnet-base-v2

- **LLaMA-based Embedders**
  Models derived from LLaMA:
  - LLaMA 2 embedder
  - Instructor-xl

4. Importance of using FAISS or chroma DB.

A vector database provides:

- Efficient nearest-neighbor search (ANN or exact) for millions of vectors.
- Several indexing options to trade accuracy vs. speed/memory.
- Persistence and APIs to save/load indexes.

FAISS specifics:

- Written in C++ with Python bindings — optimized (SIMD/multi-threading, GPU support).
- Index types: IndexFlatL2 (exact L2), IndexFlatIP (inner product), IVF, HNSW, Product Quantization (PQ) for compression.
- IndexIDMap allows storing explicit integer IDs associated with vectors (good for metadata mapping).

Why not store embeddings in Python lists?

- Lists force brute-force O(N) search, are slow at scale, and cannot utilize optimized BLAS/GPU.
- FAISS is optimized and offers orders-of-magnitude speedups (demonstrated by your demo script).

5. Chunking — why and how?

You should split large documents into semantically-coherent chunks because:

- embeddings degrade on very long texts due to truncation and diluted signal,
- retrieval becomes more precise when chunks align with single ideas or paragraphs,

Design choices I used:

- Sentence-aware chunking via NLTK sent_tokenize.
- A max_tokens threshold (approximate word count) to control chunk size.
- Fallback regex sentence splitter if NLTK not available.

Learnings from week 2:

1. **FAISS Internals**

- IndexFlatL2: exact Euclidean search.
- IndexFlatIP: exact inner-product search (normalize vectors for cosine).
- IndexIDMap: attaches your own IDs so you can map to metadata.
- Normalization is important if using cosine similarity.

2. **Persistence & Metadata Mapping**

- Must store:
  (vectors + FAISS index + metadata mapping)
- FAISS only stores vectors; you must maintain your own metadata (.json file).

3. **Resilience & Edge Cases**

- Handle empty files gracefully.
- Extremely long sentences require fallback splitting.
- Tokenization errors require unicode cleaning and try-catch logic.

4. **Importance of using FAISS or chroma DB.**

What FAISS (Facebook AI Similarity Search)?

A high-performance library from Meta for **fast vector similarity search**.

Why FAISS is powerful?

- Supports **Approximate Nearest Neighbor (ANN)** indexing
- Uses GPU acceleration
- Can handle billions of vectors
- Extremely fast search

When to use FAISS

- When scaling to millions+ embeddings
- When you need performance + accuracy
- Enterprise-level semantic search
- RAG apps with large document sets

What Is Chroma?

A lightweight, developer-friendly vector database built for LLM apps.

Why Use Chroma?

- Very easy to integrate with Python + LangChain
- Supports persistent storage
- Automatic metadata storage
- Useful for local, small to medium RAG applications

➢ **Why Not Use Python Lists or SQL Databases?**

If you store embeddings in a normal list:

- Each search would be **O(N)** time
- Slow when N = 1 million documents
- Scaling is impossible
- No indexing or optimization
- No support for Approximate Nearest Neighbor (ANN) search

Common pitfalls & how I handled them

Problem	**Symptom**	**Fix implemented**
Mismatch between return types (string vs arrays)	AttributeError: 'str' object has no attribute 'shape'	build_faiss_index() now returns (vectors, metadata, ids) and saves .npy files
Duplicate chunk IDs across multiple files	Metadata had repeated chunk IDs	Use a **global incrementing unique_id** across files in main.py
Wrong pooling (includes padding tokens)	Lower-quality embeddings	mean_pooling uses attention mask to ignore padding tokens
FAISS reconstruct() error	reconstruct not implemented for this index	Avoided reconstruct; instead save vectors to .npy for reliable reloading
Cosine vs inner-product confusion	Different score scales	**L2-normalize** embeddings and use IndexFlatIP so FAISS dot-product equals cosine

---

When running python main.py, the following sample output seen is:

Building FAISS index with 168 vectors of dimension 384.

Stored 168 embeddings in FAISS and saved metadata (visa_metadata.json).

Stored 168 embeddings in FAISS.....

--- Summary ---

Total embeddings stored: 168

Embedding dimension: 384

Metadata count: 168

Vector matrix shape: (168, 384)

Vector matrix shape: (168, 384)

--- EMBEDDING INFORMATION ---

Type: <class 'numpy.ndarray'>

Vector length: 384

Shape: (384,)

First 10 values: [-0.00229143 0.0240835 -0.06441846 -0.0180978 -0.09835444 0.06435121
 -0.03084161 0.1198636 -0.04388531 0.02947418]

Sample metadata entry for first id:

{'source': 'Canada.pdf', 'chunk_id': 0}--- Summary ---

Total embeddings stored: 168

Embedding dimension: 384

Metadata count: 168

Vector matrix shape: (168, 384)

--- EMBEDDING INFORMATION ---

Type: <class 'numpy.ndarray'>

Vector length: 384

Shape: (384,)

First 10 values: [... 10 floats ...]

Sample metadata entry for first id:

{'source': 'Canada.pdf', 'chunk_id': 0}

When running python demonstration_inner_product.py:

=== DEMO: FAISS Inner Product vs Lists (Cosine Similarity) ===

Vector dimension: 384

... sample embedding ...

FAISS Search Results:

IDs: [0 1590 3367]

Scores: [1.0 0.79 0.78]

FAISS search time: 0.00...

Brute-force List Results: ...

FAISS is ~X.XX× faster than brute-force lists!