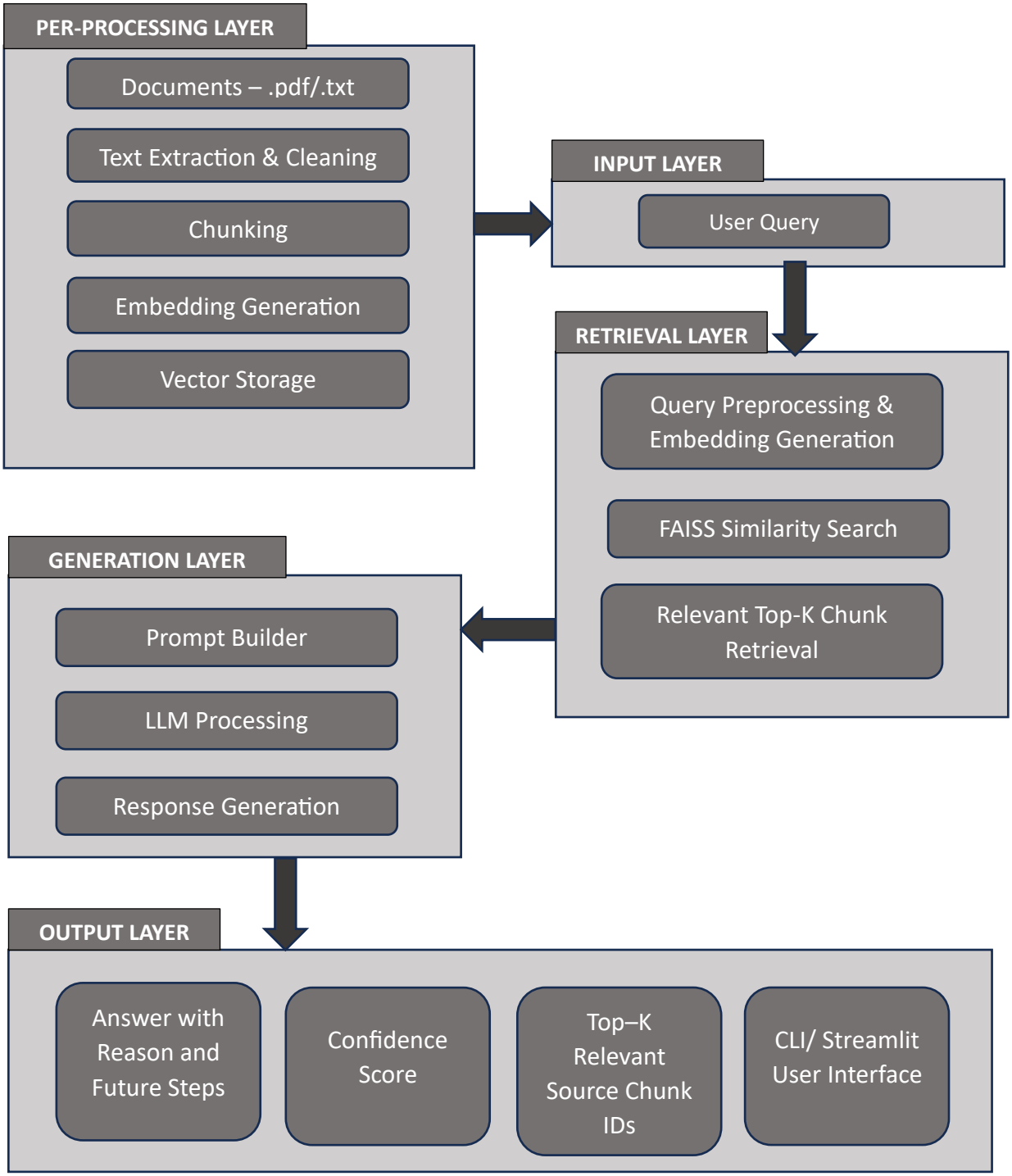
FINAL REPORT

Project Title: *SwiftVisaAI* – RAG Based Visa Eligibility Screening Agent

Description:

SwiftVisaAI is an intelligent **Retrieval-Augmented Generation (RAG)** based system designed to streamline visa eligibility screening. It enables users to query their visa eligibility for five major destinations: **United States, United Kingdom, Canada, Schengen Area, and Ireland**. By leveraging stored visa policy documents and advanced retrieval techniques, the agent provides personalized, explainable, and structured outputs to guide applicants.

Pipeline:



Preprocessing Layer:

The **Preprocessing Layer** is responsible for preparing raw visa documents (PDF/TXT) for downstream retrieval and reasoning tasks. This layer ensures that documents are cleaned, chunked, embedded, and stored efficiently in a vector database (FAISS). It forms the foundation of the Retrieval-Augmented Generation (RAG) pipeline.

1. **Input Data**

- **Folder:** Data/
- **Contents:** Raw visa documents in PDF/TXT format, along with metadata and embeddings.
- **Purpose:** Serves as the source repository for all visa-related documents that need to be processed.

2. **Utility Scripts (utils/)**

a. ***pdf_utils.py***

- **Role:** Extracts text from PDF/TXT visa documents and performs necessary preprocessing (cleaning, normalization).
- **Key Functions:**
- Text extraction from multi-page PDFs.
- Removal of unwanted characters, headers, and footers.
- Conversion into plain text for downstream processing.

👉 ***Pseudocode/Code Snippet***:

MODULE pdf_utils

```
IMPORT libraries:
    pdfplumber (for PDF text extraction)
    re (for regex text cleaning)
    logging (for suppressing noisy logs)
    Path (for file handling)
    pytesseract (for OCR)
    convert_from_path (for converting PDF pages to images)

SET logging levels for pdfminer to ERROR
------------------------------------------------------------
FUNCTION ocr_pdf(pdf_path):
    PURPOSE: Perform OCR on entire PDF if normal extraction fails
```

```
    INITIALIZE empty string text

    TRY:
        Convert PDF pages to images (high resolution)
    CATCH error:
        PRINT "PDF2Image failed"
        RETURN empty string

    PRINT "Running OCR on scanned PDF"

    FOR each image in converted pages:
        TRY:
            Extract text from image using OCR
            Append text to result
        CATCH error:
            PRINT "OCR failed on this page"

    RETURN combined text
------------------------------------------------------------
FUNCTION extract_text_from_pdf(pdf_path):
    PURPOSE: Extract text using pdfplumber, fallback to OCR if needed

    INITIALIZE empty string text

    TRY:
        OPEN PDF with pdfplumber
        FOR each page in PDF:
            EXTRACT text from page

            IF text exists:
                Append to result
            ELSE:
                TRY:
                    Convert page to image
```

```
            Run OCR on image
            Append OCR text
        CATCH error:
            PRINT "Page-level OCR failed, fallback to full OCR"
            RETURN ocr_pdf(pdf_path)
    CATCH error:
        PRINT "pdfplumber failed, fallback to full OCR"
        RETURN ocr_pdf(pdf_path)

    IF extracted text is empty:
        PRINT "No text extracted, fallback to full OCR"
        RETURN ocr_pdf(pdf_path)

    RETURN extracted text
------------------------------------------------------------
FUNCTION read_text_file(txt_path):
    PURPOSE: Safely read a text file

    CREATE Path object from txt_path
    READ file with UTF-8 encoding, ignoring errors
    RETURN file content
------------------------------------------------------------
FUNCTION clean_text(text):
    PURPOSE: Normalize and clean extracted text

    REPLACE unwanted control characters with newline
    LIMIT consecutive newlines to maximum of 2
    REPLACE multiple spaces/tabs with single space
    REPLACE bullet symbols (•, ) with "-"
    STRIP leading/trailing spaces

    RETURN cleaned text
END MODULE
```

b. ***chunking.py***

- **Role:** Splits large visa documents into smaller, manageable text chunks.
- **Importance:** Chunking ensures that embeddings capture localized context and improves retrieval accuracy.
- **Key Functions:**
- Sentence/paragraph segmentation.
- Fixed-size chunk creation (e.g., 500 tokens).

👉 ***Pseudocode/Code Snippet:***

MODULE chunking

IMPORT sentence tokenizer from NLTK

IMPORT regex library

FUNCTION _fallback_sent_tokenize(text):

PURPOSE: Provide a backup sentence splitter if NLTK fails

STRIP leading/trailing spaces from text

SPLIT text into sentences using regex:

- Look for punctuation marks (., !, ?) followed by whitespace

FILTER out empty strings

RETURN list of sentences

FUNCTION sentence_chunking(text, max_tokens = 500):

PURPOSE: Split text into chunks, each with at most `max_tokens` words

TRY:

USE NLTK sent_tokenize to split text into sentences

CATCH error:

USE _fallback_sent_tokenize to split text

INITIALIZE empty list chunks

INITIALIZE empty list current_chunk

INITIALIZE token_count = 0

FOR each sentence in sentences:

SPLIT sentence into tokens (words)

```
        IF number of tokens > max_tokens:
            IF current_chunk is not empty:
                ADD current_chunk (joined as string) to chunks
                RESET current_chunk and token_count

            SPLIT the long sentence into slices of size max_tokens
            FOR each slice:
                ADD slice (joined as string) to chunks
            CONTINUE to next sentence

        IF token_count + number of tokens > max_tokens:
            ADD current_chunk (joined as string) to chunks
            RESET current_chunk and token_count

        ADD sentence to current_chunk
        INCREMENT token_count by number of tokens

    IF current_chunk is not empty:
        ADD current_chunk (joined as string) to chunks

    RETURN chunks (list of strings)
------------------------------------------------------------
END MODULE
```

c. ***embedding.py***

- **Role:** Converts text chunks into numerical embeddings of **384 dimensions**.
- **Importance:** Embeddings allow semantic similarity search across visa documents.

Key Functions:

- Integration with sentence-transformer models.
- Generation of dense vector representations.

👉 ***Pseudocode/Code Snippet***:

```
MODULE embedding


IMPORT torch (for tensor operations)
IMPORT numpy (for numerical arrays)
IMPORT AutoTokenizer, AutoModel (from transformers library)
------------------------------------------------------------
SET MODEL_NAME = "sentence-transformers/all-MiniLM-L6-v2"


INITIALIZE tokenizer using AutoTokenizer with MODEL_NAME
INITIALIZE model using AutoModel with MODEL_NAME
SET model to evaluation mode (no training)
------------------------------------------------------------
FUNCTION mean_pooling(outputs, attention_mask):
    PURPOSE: Compute attention-aware mean pooling of token embeddings


    token_embeddings ← outputs.last_hidden_state   # shape: (batch_size, seq_len, hidden)
    mask ← expand attention_mask to shape (batch_size, seq_len, 1)
    summed ← sum of (token_embeddings * mask) along sequence dimension
    counts ← sum of mask along sequence dimension, clamp minimum to avoid division by zero
    RETURN (summed / counts)   # average embedding per sequence
------------------------------------------------------------
FUNCTION get_embedding(text_chunk):
    PURPOSE: Generate normalized embedding vector for given text


    Tokenize text_chunk into model inputs (with truncation + padding)
    Disable gradient computation
    Pass inputs through model → outputs


    emb ← mean_pooling(outputs, attention_mask)
    Convert emb to numpy array
    Cast emb to float32
```

```
    norm ← L2 norm of emb
    IF norm > 0:
        Normalize emb by dividing by norm

    RETURN emb (normalized vector)
------------------------------------------------------------
FUNCTION print_embedding_info(embedding):
    PURPOSE: Print diagnostic information about embedding

    PRINT "Type of embedding"
    PRINT "Vector length"
    PRINT "Shape of embedding"
    PRINT "First 10 values of embedding"
------------------------------------------------------------
END MODULE
```

d. ***vector_store.py***

- **Role:** Stores embeddings into a **FAISS vector database** for efficient similarity search.
- **Key Functions:**
- Index creation and persistence.
- Querying nearest neighbors for retrieval.
- Updating the store with new documents.

👉 ***Pseudocode/Code Snippet:***

```
MODULE vector_store

IMPORT os
IMPORT json
IMPORT numpy
IMPORT faiss
------------------------------------------------------------
FUNCTION build_faiss_index(
    embeddings,
    index_path = "visa_embeddings.index",
```

```
    metadata_path = "visa_metadata.json",
    vectors_npy = "visa_embeddings.npy",
    ids_npy = "visa_ids.npy"
):
PURPOSE: Build and save a FAISS index from sentence embeddings

INPUT:
    embeddings → list of dicts {unique_id, chunk_id, source, embedding}
    index_path → file path to save FAISS index
    metadata_path → file path to save metadata JSON
    vectors_npy → file path to save raw vectors
    ids_npy → file path to save vector IDs

PROCESS:
    - Convert embeddings list into numpy array of vectors (float32)
    - Convert unique_ids into numpy array of IDs (int64)
    - Determine embedding dimension
    - PRINT number of vectors and dimension

    - Normalize vectors (L2 norm) for cosine similarity
    - Create FAISS inner-product index
    - Wrap index with ID mapping (so queries return IDs)
    - Add vectors with IDs to index

    - Save FAISS index to disk
    - Save vectors and IDs as .npy files

    - Build metadata dictionary:
        For each embedding:
            key = unique_id
            value = {source, chunk_id}
    - Save metadata dictionary as JSON file

    - PRINT confirmation of saved index and metadata
```

```
    OUTPUT:
        RETURN vectors, metadata, ids
------------------------------------------------------------
FUNCTION load_faiss_index(index_path):
    PURPOSE: Load FAISS index from file

    INPUT:
        index_path → path to saved FAISS index
    PROCESS:
        - Read FAISS index from file
    OUTPUT:
        RETURN loaded FAISS index
------------------------------------------------------------
END MODULE
```

3. **Workflow Integration** (***main.py***)

- **Role:** Orchestrates the preprocessing pipeline by calling the utility scripts in sequence.
- **Steps:**

1. Load raw visa documents from Data/.
2. Extract text using pdf_utils.py.
3. Chunk text with chunking.py.
4. Generate embeddings via embedding.py.
5. Store embeddings in FAISS using vector_store.py.

👉 ***Pseudocode/Code Snippet***:

```
MODULE main

IMPORT required libraries:
    os, json, Path (for file handling)
    ensure_nltk_resources (to set up NLTK)
    extract_text_from_pdf, read_text_file, clean_text (PDF/TXT utilities)
    sentence_chunking (for splitting text into chunks)
    get_embedding, print_embedding_info (for embeddings)
    build_faiss_index (for FAISS vector store)
```

```
CALL ensure_nltk_resources() to make sure NLTK is ready


SET DATA_DIR = "Data"
SET PDF_DIR = "Data/pdfs"
------------------------------------------------------------
FUNCTION process_documents():
   PURPOSE: Process all PDF/TXT files in Data/pdfs
         → extract text
         → clean text
         → chunk into sentences
         → embed chunks
         → save chunks JSON
         → return embeddings list

   INITIALIZE empty list all_embeddings
   INITIALIZE empty dictionary chunks_json
   INITIALIZE uid = 0

   FOR each file in PDF_DIR (sorted order):
      IF file extension is not .pdf or .txt:
         SKIP file

      PRINT "Processing file"

      IF file is PDF:
         text ← extract_text_from_pdf(file)
      ELSE:
         text ← read_text_file(file)

      text ← clean_text(text)

      IF text is empty:
         PRINT "Empty file, skipping"
```

```
            CONTINUE

        chunks ← sentence_chunking(text)

        FOR each chunk in chunks:
            emb ← get_embedding(chunk)

            SAVE chunk text in chunks_json with key = uid
            APPEND embedding record to all_embeddings:
                {
                    unique_id: uid,
                    chunk_id: uid,
                    source: file name,
                    embedding: emb
                }

            INCREMENT uid

    SAVE chunks_json to "Data/visa_chunks.json" as JSON

    PRINT "Total chunks saved"
    RETURN all_embeddings
------------------------------------------------------------
MAIN ENTRY POINT:
IF script is run directly:

    PRINT "STEP 1 — PROCESSING DOCUMENTS"
    embeddings ← process_documents()

    IF embeddings is empty:
        PRINT "No embeddings generated, check PDFs"
        EXIT program

    PRINT "STEP 2 — BUILDING FAISS INDEX"
```

```
    vecs, meta, ids ← build_faiss_index(
        embeddings,
        index_path = "Data/visa_embeddings.index",
        metadata_path = "Data/visa_metadata.json",
        vectors_npy = "Data/visa_embeddings.npy",
        ids_npy = "Data/visa_ids.npy"
    )

    PRINT "Stored number of embeddings"
    CALL print_embedding_info on first embedding vector
------------------------------------------------------------
END MODULE
```

4. **Output**

- **Vector Database:** FAISS index containing embeddings of visa documents.
- **Logs:** Processing decisions are recorded in logs/decision_log.jsonl.
- **Usage:** The preprocessed data is later consumed by the RAG pipeline

Input Layer

The **Input Layer** is the entry point for user interaction with the SwiftVisa AI system. It captures visa-related queries from users, processes them into a structured format, and passes them into the Retrieval-Augmented Generation (RAG) pipeline.

---

1. User Query Interfaces

a. ***query_cli.py***

- **Role:** Provides a **command-line interface (CLI)** for users to input visa-related questions.
- **Key Functions:**
    - Accepts text queries from the terminal.
    - Sends queries to the RAG pipeline for processing.
    - Displays answers directly in the CLI.

👉 ***Pseudocode/Code Snippet****:*

```
LOAD environment variables

SET TOP_K_DISPLAY = 5


FUNCTION format_llm_response(parsed):

  DEFINE normalize(value):

    - Handle None → "Not Provided"

    - Handle list → filter out "null"/"not provided", format items

    - Handle dict → format key:value pairs

    - Else → string


  GET decision, reason from parsed

  IF neither exists:

    RETURN raw response or fallback message


  RETURN formatted string with:

    - Eligibility Status

    - Confidence Score

    - Reason for Decision

    - Actions to Improve
```

```
FUNCTION main():
    PRINT intro message

    LOOP until user types "exit" or "quit":
        ASK user for visa question
        CLEAN input

        CALL run_rag(query, top_k=TOP_K_DISPLAY)
        parsed ← result["parsed"]

        PRINT formatted LLM response
        PRINT top relevant chunk UIDs (if any)
        PRINT final blended confidence score

    PRINT "Goodbye"

IF run as script:
    CALL main()
```

b. ***streamlit_app.py***

- **Role:** Offers a **web-based frontend** for user queries using Streamlit.
- **Key Functions:**
    - Interactive UI for entering visa questions.
    - Displays retrieved answers and logs.
    - Provides a user-friendly interface for non-technical users.

👉 ***Pseudocode/Code Snippet:***

```
SETUP:
    - Load environment variables
    - Configure Streamlit page (title, layout, sidebar)
    - Apply custom CSS styling

UTILITY:
```

FUNCTION format_llm_response_streamlit(parsed):

Normalize values (None, list, dict)

If no structured info → show raw fallback

Else → return formatted HTML with:

- Eligibility Status
- Confidence Score
- Reason for Decision
- Actions to Improve

COMPONENTS:

FUNCTION render_hero_section():

Show flag banner + title

Display feature cards (Instant Analysis, Accurate Results, Multiple Countries)

FUNCTION live_query_tab():

Show query input area + analyze button

On button click:

- Run RAG pipeline with query
- Display structured LLM response
- Show metrics (confidence %, docs analyzed)
- Show referenced documents as chips

Handle errors gracefully

FUNCTION render_sidebar():

Sidebar with quick guide:

- How to use
- Supported visa types
- Countries covered
- Tips + disclaimer
- Contact info

MAIN:

FUNCTION main():

Render sidebar, hero section, live query tab

```
    Show footer with credits


ENTRY:
  IF run as script:
    TRY run main()
    CATCH startup errors → print critical failure message
```

c. *user_queries.json*

- **Role:** Stores user queries for batch processing or testing.
- **Usage:** Acts as a dataset of real or simulated queries for evaluation.

---

d. *process_test_queries.py*

- **Role:** Automates the processing of queries from user_queries.json.
- **Key Functions:**
  - Reads queries in bulk.
  - Passes them through the RAG pipeline.
  - Stores results in query_results.json.

👉 ***Pseudocode/Code Snippet:***

```
SETUP:
  - Load environment variables (API keys, etc.)


FUNCTION process_queries(query_file="user_queries.json", output_file="query_results.json"):
  IF query_file does not exist:
    PRINT error and return


  READ queries from query_file (JSON list)


  INIT empty results list
  PRINT number of queries


  FOR each query in queries:
    GET query text
    GET optional user_profile (for logging only)
```

```
    PRINT progress message

    TRY:
      result ← run_rag(query)
      APPEND {query, user_profile, response=result} to results
    EXCEPT error:
      PRINT error message
      APPEND {query, user_profile, response={"error": error}} to results

  WRITE results to output_file (JSON, pretty format, allow non-ASCII)
  PRINT completion message

MAIN:
  IF run as script:
    CALL process_queries()
```

2. <u>Workflow</u>

1. User enters a query via CLI (query_cli.py) or Streamlit (streamlit_app.py).
2. Queries are optionally logged in user_queries.json.
3. Batch queries are processed using process_test_queries.py.
4. Results are stored in query_results.json for analysis.

Retrieval Layer

The **Retrieval Layer** is responsible for fetching the most relevant chunks of visa documents from the FAISS vector database. It ensures that the system provides contextually accurate information to the LLM for generating answers.

---

1. <u>Core Retrieval Components (rag/)</u>

a. *retriever.py*

- **Role:** Handles FAISS-based similarity search.
- **Key Functions:**
    - Loads FAISS index created during preprocessing.
    - Retrieves top-k most relevant chunks based on query embeddings.
    - Returns results with metadata for logging and prompt construction.

👉 ***<u>Pseudocode/Code Snippet:</u>***

```
SETUP:
  Define paths for FAISS index, metadata JSON, chunks JSON, and IDs file

CLASS Retriever:
  INIT(index_path):
    - Load FAISS index from file
    - Load metadata JSON (or empty dict)
    - Load chunks JSON (or empty dict)
    - Load IDs array (or None)
    - Define keyword lists (financial, sponsorship, study)

  FUNCTION _keyword_match_score(text, keywords):
    - Lowercase text
    - Count keyword matches
    - Return score capped at 1.0 (3+ hits = 1.0)

  FUNCTION retrieve(query, top_k=5, query_embedding=None):
    INIT empty results list
    Extract keywords from query
```

IF query_embedding provided:

- Normalize embedding
- Search FAISS index for top_k*3 results
- For each result:
  - Get metadata + chunk text
  - Compute keyword score
  - Combine scores (60% vector, 40% keyword)
  - Append result dict

ELSE:

- Loop through chunks
- Compute keyword score
- Append results if score > 0

Sort results by score (descending)

Remove duplicates by UID

Keep top_k results

IF fewer than top_k:

- Add fallback chunks with score=0 until top_k reached

RETURN results

b. ***pipeline.py***

- **Role:** Orchestrates the RAG pipeline, including retrieval.
- **Key Functions:**
  - Accepts user query embeddings.
  - Calls retriever.py to fetch relevant chunks.
  - Passes retrieved chunks to the LLM client for answer generation.

👉 ***Pseudocode/Code Snippet:***

```
SETUP:
    - Import dependencies (json, re, numpy, SentenceTransformer, Retriever, prompt_builder, llm_client, logger)
    - Initialize Retriever
    - Try to load SentenceTransformer model (384-dim); fallback to None if fails

FUNCTION get_embedding(text):
    IF model unavailable → return zero vector
    ELSE → encode text to 384-dim embedding
    RETURN embedding (list of floats)

FUNCTION compute_confidence_from_scores(scores):
    IF empty → return 0.0
    ELSE → clip scores [-1,1], map to [0,1], return mean

FUNCTION extract_info(text):
    INIT result dict with default fields (decision, confidence, reason, future_steps, raw)
    IF text empty or error → set decision="ERROR", reason=error message
    ELSE TRY parse text as JSON:
        - Map keys to result fields
        - Normalize decision, confidence, future_steps
        - Remove null/invalid values
        RETURN result
    IF JSON parsing fails:
        - Use raw text as reason
        - Regex match eligibility keywords for decision
    RETURN result

FUNCTION run_rag(query, top_k=5):
    query_embedding ← get_embedding(query)
    retrieved ← retriever.retrieve(query, top_k, query_embedding)
    scores ← list of retrieval scores
    retrieval_conf ← compute_confidence_from_scores(scores)
```

```
prompt ← build_prompt(query, retrieved, max_chars=3000, mode="decision")

TRY raw_resp ← call_gemini(prompt)
EXCEPT → raw_resp = error message

parsed ← extract_info(raw_resp)

declared_conf ← parsed.confidence or 0.0
final_conf ← 0.6*retrieval_conf + 0.4*declared_conf (fallback to retrieval_conf if error)

yes_no ← map decision string:
   "eligible" → 1
   "not eligible" → 0
   "partially eligible" → 0.5

answer_obj ← {
   parsed, final_confidence, retrieval_score_mean, retrieved, raw_llm, yes_no
}

TRY log_decision(query, retrieved, answer_obj, raw_prompt=prompt)
EXCEPT print error
RETURN answer_obj
```

c. ***logger.py***

- **Role:** Logs retrieval decisions and query results.
- **Key Functions:**
    - Records which chunks were retrieved.
    - Stores query-answer pairs in logs/decision_log.jsonl.

👉 ***Pseudocode/Code Snippet:***

```
SETUP:
   - Create "logs" directory if not exists
   - Define file paths:
      DECISION_LOG_FILE = logs/decision_log.jsonl
      CONVERSATION_LOG_FILE = logs/conversation_log.jsonl


FUNCTION log_decision(query, retrieved, answer_json, raw_prompt=None):
   TRY:
      entry = {
         timestamp (UTC ISO),
         query,
         retrieved_count,
         retrieved → list of {uid, score, source},
         decision,
         confidence,
         final_confidence,
         raw_prompt (boolean flag)
      }
      APPEND entry as JSON line to DECISION_LOG_FILE
   EXCEPT error:
      PRINT failure message
FUNCTION log_conversation(profile_key, role, text, metadata=None):
   TRY:
      entry = {
         timestamp (UTC ISO),
         profile_key,
         role (user/assistant),
         text (truncated to 500 chars),
         metadata (dict or empty)
      }
      APPEND entry as JSON line to CONVERSATION_LOG_FILE
   EXCEPT error:
      PRINT failure message
```

2. Workflow

1. Query embedding is generated in the Input Layer.
2. retriever.py searches FAISS for the most relevant chunks.
3. Retrieved chunks are logged (logger.py) and passed to pipeline.py.
4. The LLM uses these chunks to generate a final answer.

---

3. Output

- **Relevant Chunks:** Retrieved text segments from visa documents.
- **Logs:** Stored in logs/decision_log.jsonl for transparency and debugging.
- **Usage:** Provides contextual grounding for the LLM to answer visa queries accurately.

Generation Layer

The **Generation Layer** is responsible for constructing prompts, sending them to the Large Language Model (LLM), and receiving responses. It combines the user query with retrieved context chunks to ensure the LLM produces grounded, accurate answers.

---

1. Core Components (rag/)

a. ***prompt_builder.py***

- **Role:** Builds structured prompts by combining the user query with retrieved chunks.

  Key Functions:

  - Formats query + context into a single prompt.
  - Ensures consistency in instructions for the LLM.
  - Adds eligibility reasoning templates.

👉 ***Pseudocode/Code Snippet:***

```
FUNCTION build_prompt(query, retrieved, max_chars=3000, mode="decision"):
  INIT ctx_parts = [], total = 0

  FOR each retrieved chunk (with index):
    GET metadata (chunk_id, source, uid)
    CREATE header string with source + chunk_id
    GET text content
    piece = header + text
```

```
    IF adding piece exceeds max_chars:
        ADD truncated piece (up to remaining chars)
        BREAK loop
    ELSE:
        ADD piece to ctx_parts
        INCREMENT total

IF ctx_parts not empty:
    context_text = join pieces with "---"
ELSE:
    context_text = "No context was retrieved. Use NULL where needed."

CREATE final prompt string:
    - Role: SwiftVisa immigration assistant
    - Context: context_text
    - User Question: query
    - Instructions:
        • Output JSON only
        • Decision = eligible / not eligible / partially eligible
        • Confidence = 0.0–1.0
        • Use only provided context
        • Reason concise (2–4 sentences)
    - Output format: JSON object with fields:
        eligibility_status, reason, future_steps, confidence

RETURN prompt
```

b. ***llm_client.py***

- **Role:** Provides wrappers for LLM APIs (Gemini, OpenAI).
- **Key Functions:**
    - Sends prompts to the chosen LLM.
    - Receives and parses responses.
    - Handles fallback between local and API-based models.

👉 ***Pseudocode/Code Snippet***:

```
SETUP:
   - Load environment variables
   - Get GEMINI_API_KEY
   - Configure genai with API key
   - Set DEFAULT_MODEL = "models/gemini-2.5-flash"


FUNCTION _extract_text_from_response(resp):
   IF resp is None → return ""
   TRY resp.text → return if non-empty
   TRY resp.prompt_feedback → return block_reason or feedback info
   TRY resp.candidates → collect text parts from candidates
   RETURN combined text or "" if nothing found


FUNCTION _simplify_prompt_for_retry(original_prompt):
   RETURN simplified fallback prompt:
      - Short factual JSON answer
      - Must include eligibility_status, reason, future_steps, confidence
      - No markdown wrappers


FUNCTION call_gemini(prompt, model_name=DEFAULT_MODEL, max_output_tokens=1024, temperature=0.0,
retry_on_empty=True):
   IF GEMINI_API_KEY missing → return error


   TRY initialize model
   EXCEPT → return error


   DEFINE _call_once(p):
      TRY model.generate_content(p, config={max_output_tokens, temperature})
      EXCEPT → return error string
      text ← _extract_text_from_response(resp)
      IF text empty → check prompt_feedback for block_reason
      RETURN (text, resp)
```

```
    # First attempt
    text, raw ← _call_once(prompt)
    IF text valid (not error) → return text

    # Retry with simplified prompt if enabled
    IF retry_on_empty:
        simple ← _simplify_prompt_for_retry(prompt)
        text2, raw2 ← _call_once(simple)
        IF text2 valid → return text2
        ELSE IF text2 non-empty → return text2

    # Final fallback
    IF text non-empty → return text
    RETURN "ERROR: Model output filtered or empty."
```

c. ***pipeline.py***

- **Role:** Orchestrates the full RAG pipeline, including prompt building and LLM response generation.
- **Key Functions:**
    - Accepts user query and retrieved chunks.
    - Calls prompt_builder.py to construct the final prompt.
    - Sends prompt to llm_client.py for response.
    - Passes response to the Output Layer.

👉 ***Pseudocode/Code Snippet:***

```
SETUP:
    - Import dependencies (json, re, numpy, SentenceTransformer, Retriever, prompt_builder, llm_client, logger)
    - Initialize Retriever
    - Try to load SentenceTransformer model (384-dim); fallback to None if fails

FUNCTION get_embedding(text):
    IF model unavailable → return zero vector
    ELSE → encode text to 384-dim embedding
    RETURN embedding list
```

```
FUNCTION compute_confidence_from_scores(scores):
    IF empty → return 0.0
    ELSE → clip scores [-1,1], map to [0,1], return mean

FUNCTION extract_info(text):
    INIT result dict with default fields (decision, confidence, reason, future_steps, raw)
    IF text empty or error → set decision="ERROR", reason=error message
    ELSE TRY parse text as JSON:
        - Map keys to result fields
        - Normalize decision, confidence, future_steps
        - Remove null/invalid values
        RETURN result
    IF JSON parsing fails:
        - Use raw text as reason
        - Regex match eligibility keywords for decision
    RETURN result

FUNCTION run_rag(query, top_k=5):
    query_embedding ← get_embedding(query)
    retrieved ← retriever.retrieve(query, top_k, query_embedding)
    scores ← list of retrieval scores
    retrieval_conf ← compute_confidence_from_scores(scores)

    prompt ← build_prompt(query, retrieved, max_chars=3000, mode="decision")

    TRY raw_resp ← call_gemini(prompt)
    EXCEPT → raw_resp = error message

    parsed ← extract_info(raw_resp)

    declared_conf ← parsed.confidence or 0.0
    final_conf ← 0.6*retrieval_conf + 0.4*declared_conf (fallback to retrieval_conf if error)

    yes_no ← map decision string:
```

```
    "eligible" → 1
    "not eligible" → 0
    "partially eligible" → 0.5

  answer_obj ← {
    parsed, final_confidence, retrieval_score_mean, retrieved, raw_llm, yes_no
  }

  TRY log_decision(query, retrieved, answer_obj, raw_prompt=prompt)
  EXCEPT print error

  RETURN answer_obj
```

2. <u>Workflow</u>

1. User query + top-k retrieved chunks are passed to prompt_builder.py.
2. A structured prompt is created and sent to the LLM via llm_client.py.
3. The LLM generates a response, which is returned to pipeline.py.
4. The response is forwarded to the Output Layer for display.

Output Layer

The **Output Layer** is responsible for presenting the final response to the user. It not only displays the LLM's answer but also provides transparency by showing related chunk IDs, eligibility reasoning, and future steps.

---

1. <u>Output Components</u>

a. *Response Display*

- **Role:** Shows the LLM's generated answer.
- **Files:**
    - streamlit_app.py → Web dashboard UI.
    - query_cli.py → Terminal-based CLI bot.

👉 ***<u>Pseudocode/Code Snippet</u>***:

streamlit_app.py

```
# Display structured LLM response in Streamlit
st.markdown("<h3>🗣️ AI Assistant - Your Eligibility Analysis</h3>", unsafe_allow_html=True)
st.markdown(format_llm_response_streamlit(parsed), unsafe_allow_html=True)
```

query_cli.py

```
# Display structured LLM response in CLI
print("\n--- 🗣️ SwiftVisa Assistant ---")
print(format_llm_response(parsed))
```

b. *Related Chunks*

- **Role:** Displays the **top 5 related chunk IDs** retrieved from FAISS.
- **Purpose:** Provides transparency into which document segments influenced the answer.

👉 ***<u>Pseudocode/Code Snippet</u>***:

streamlit_app.py

```
# Show referenced document chunks in Streamlit
if retrieved:
    st.markdown("<h3>📑 Referenced Documents</h3>", unsafe_allow_html=True)
    doc_chips = "".join([
```

```
        f'<span class="doc-chip">📄 Doc #{c.get("uid", "N/A")}</span>'
        for c in retrieved[:TOP_K_DISPLAY]
    ])
    st.markdown(doc_chips, unsafe_allow_html=True)
else:
    st.info("No relevant documents were retrieved.")
```

query_cli.py

```
# Show top chunk IDs in CLI
print("\n--- 📄 RELEVANT CHUNK UIDs ---")
if retrieved:
    uids = [str(c.get("uid", "No UID")) for c in retrieved[:TOP_K_DISPLAY]]
    print(f"Chunks Used (Top {TOP_K_DISPLAY} UIDs): {', '.join(uids)}")
else:
    print("No relevant documents were retrieved.")
```

c. *Eligibility Reasoning*

- **Role:** Explains why the user may or may not be eligible for a visa.
- **Purpose:** Adds interpretability to the system's decision-making.

👉 ***Pseudocode/Code Snippet***:

streamlit_app.py

```
# Display reasoning in Streamlit
st.markdown("<h3>💡 Reason for Decision</h3>", unsafe_allow_html=True)
st.markdown(f"<p>{parsed.get('reason', 'No reasoning provided.')}</p>", unsafe_allow_html=True)
```

query_cli.py

```
# Display reasoning in CLI
print(f"**Reason for Decision:** {parsed.get('reason', 'No reasoning provided.')}")
```

d. *Future Steps*

- **Role:** Suggests next actions for the user (e.g., required documents, application steps).
- **Purpose:** Provides actionable guidance beyond the immediate answer.

👉 ***Pseudocode/Code Snippet:***

streamlit_app.py

```
# Display future steps in Streamlit
st.markdown("<h3>🚀 Actions to Improve</h3>", unsafe_allow_html=True)
future_steps = parsed.get("future_steps", [])
if future_steps:
    st.markdown("\n".join([f"- {step}" for step in future_steps]))
else:
    st.info("No future steps suggested.")
```

query_cli.py

```
# Display future steps in CLI
print("**Actions to Improve:**")
future_steps = parsed.get("future_steps", [])
if future_steps:
    for step in future_steps:
        print(f"- {step}")
else:
    print("None")
```

2. Workflow

1. The LLM response is received from the Generation Layer.
2. The Output Layer displays:
    - Final answer.
    - Top 5 related chunk IDs.
    - Eligibility reasoning.
    - Suggested future steps.
3. The response is shown either in:
    - **Streamlit Dashboard** (streamlit_app.py) for interactive UI.
    - **CLI Bot** (query_cli.py) for terminal-based usage.

---

4. Output Example (Structure)

Answer: [LLM-generated response]

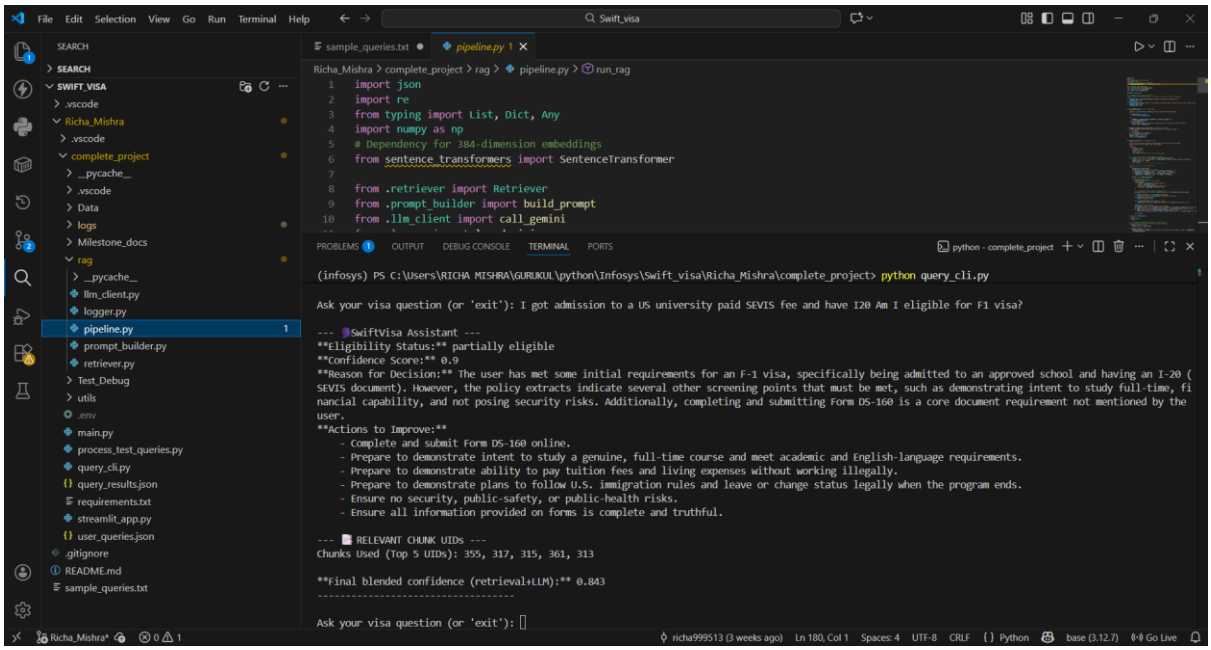
Top 5 Related Chunks: [IDs]

Eligibility Reasoning: [Explanation]

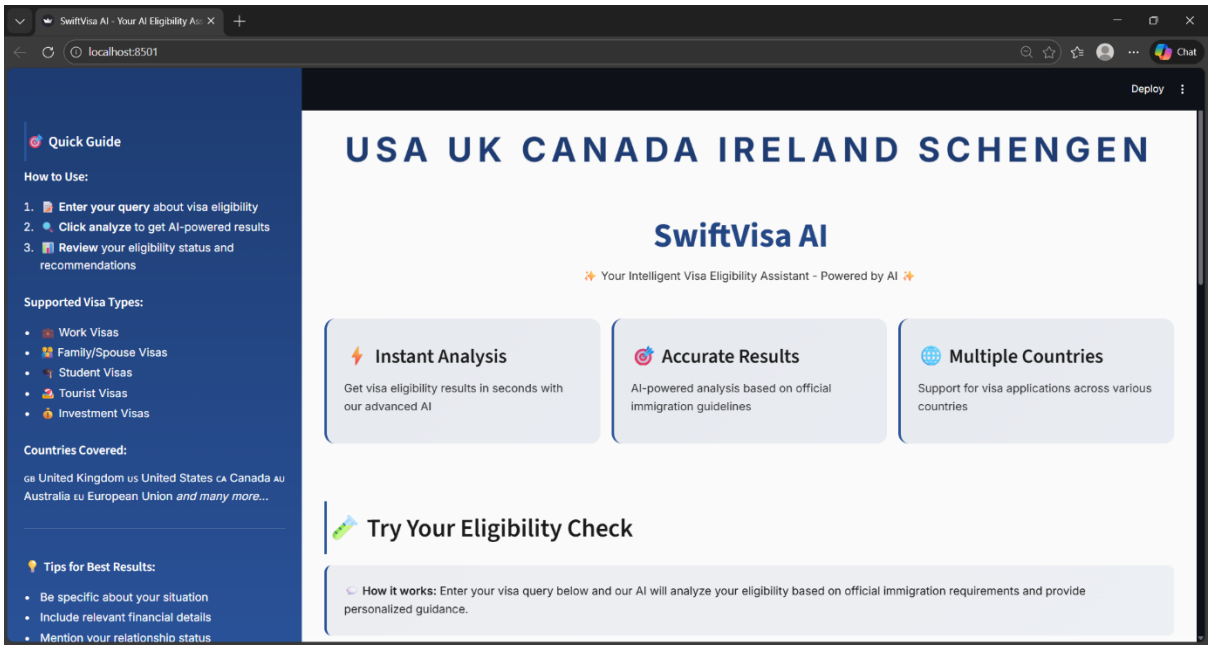
Future Steps: [Suggested actions]

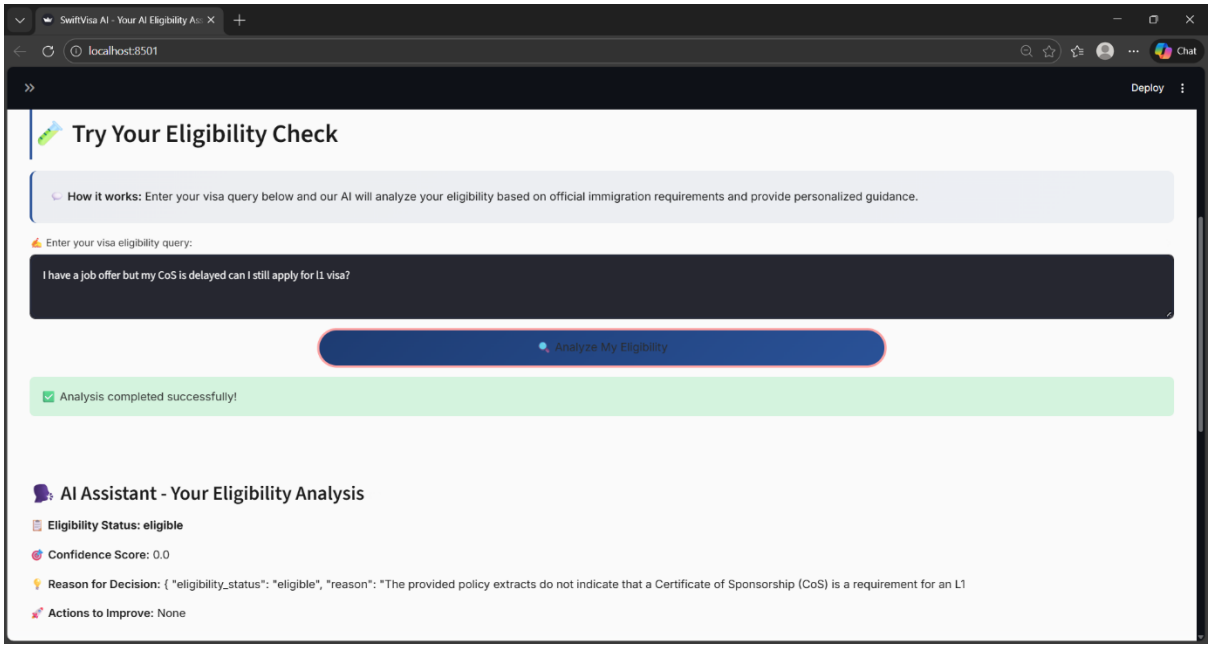
Output Screen:

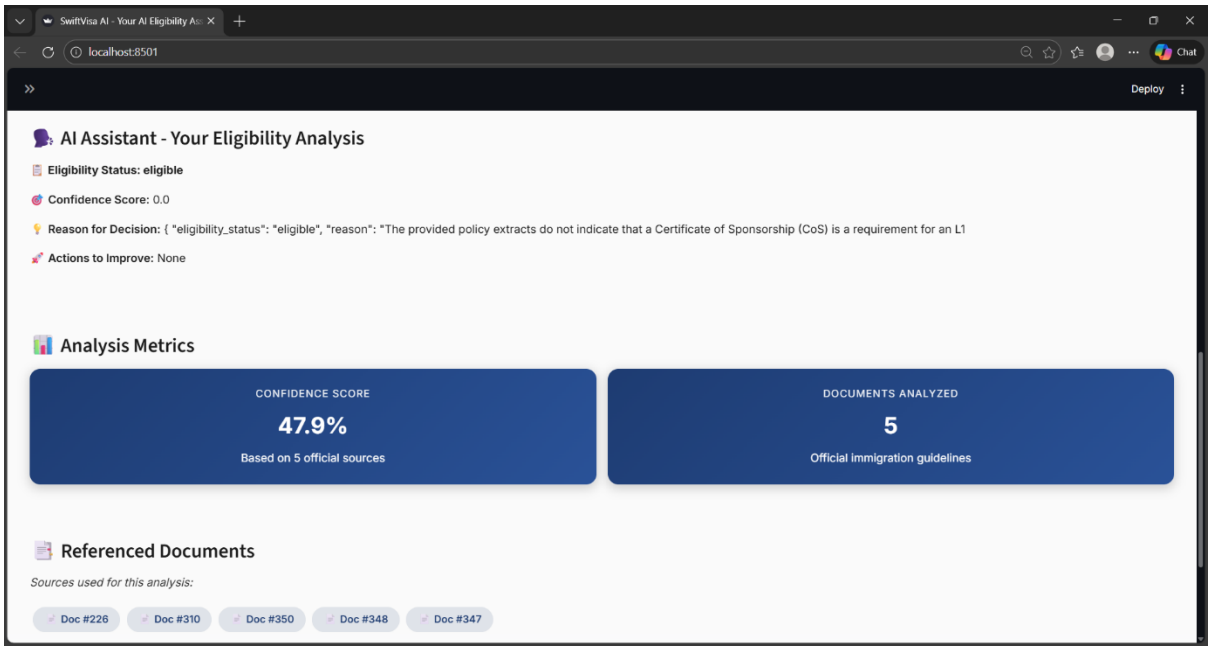
CLI Chatbot Screen:



Streamlit Dashboard Screen:







DEMO VIDEO LINK:

https://drive.google.com/file/d/11p3o2_ApnHMHn8Iq7ntKQTdXAPJJCAmD/view?usp=sharing