INTERNSHIP PROJECT REPORT

SwiftVisaAI – RAG Based Visa Eligibility Screening Agent

Submitted by:

Richa Mishra

Internship Role: **AI/ML Intern**

Project Duration: **November2025 – January 2026**

Submitted to:

Infosys

Mentored by:

Mr. Siddarth

Position: **Data Scientist**

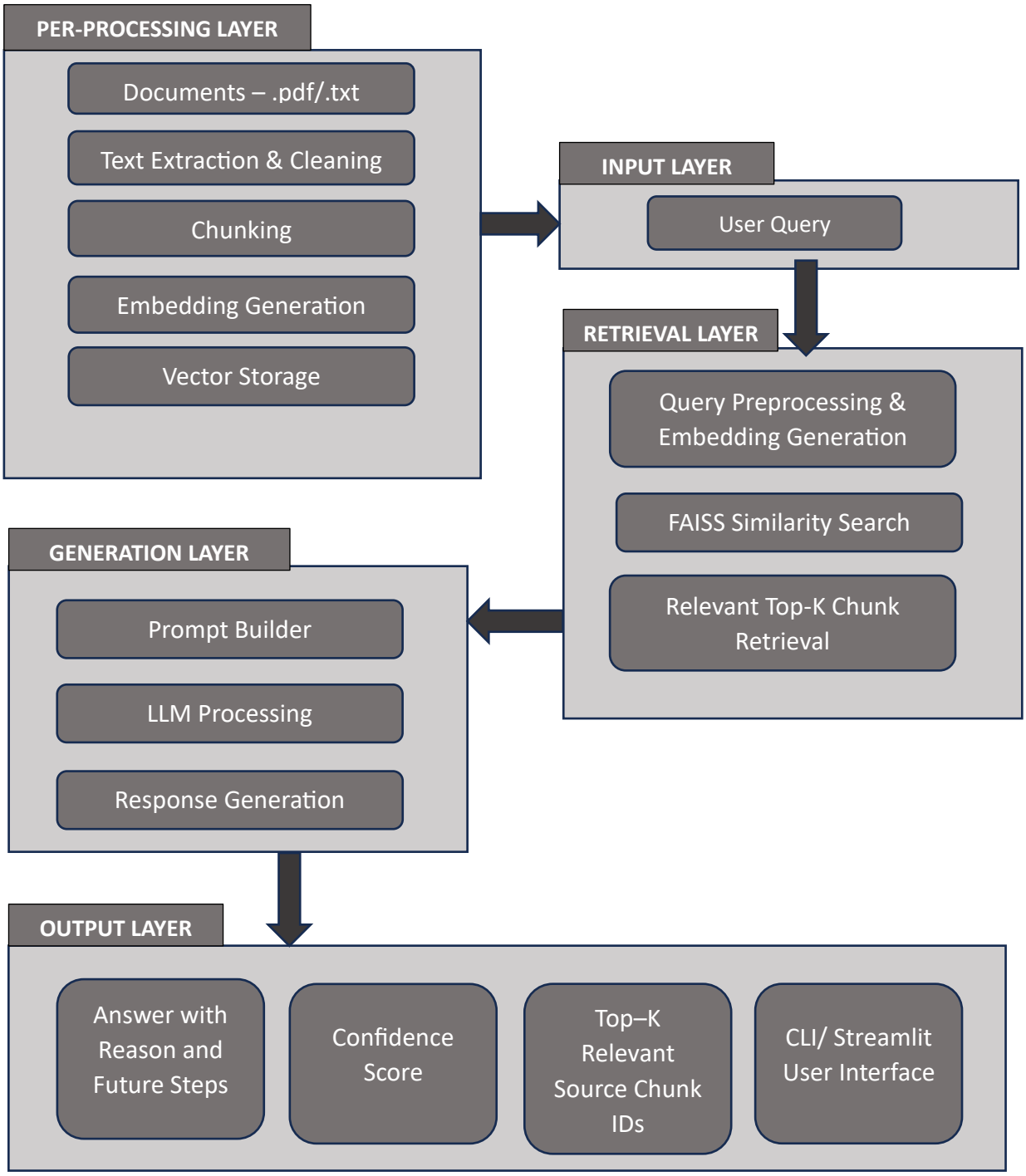
Submission Date:

December 2025

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 for five countries : USA, UK, Ireland, Canada and Schengen.
  - Data folder contains another folder pdfs/ that contains documents in .txt and .pdf format.
  - It also contains :
    - visa_chunks.json
    - visa_embeddings.index
    - visa_embeddings.npy
    - visa_metadata.json
- **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

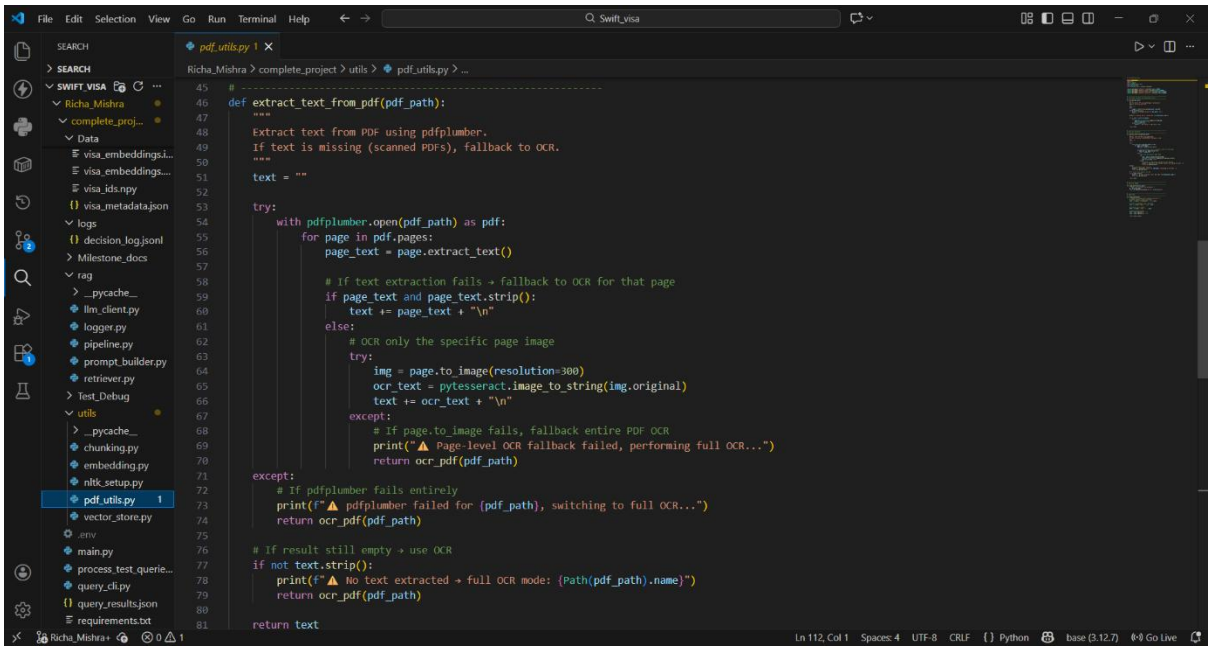
FUNCTION ocr_pdf(path): RETURN extracted_text

FUNCTION extract_text_from_pdf(path): RETURN text_or_ocr

FUNCTION read_text_file(path): RETURN cleaned_text

FUNCTION clean_text(text): RETURN normalized_text

END MODULE



b. ***chunking.py***

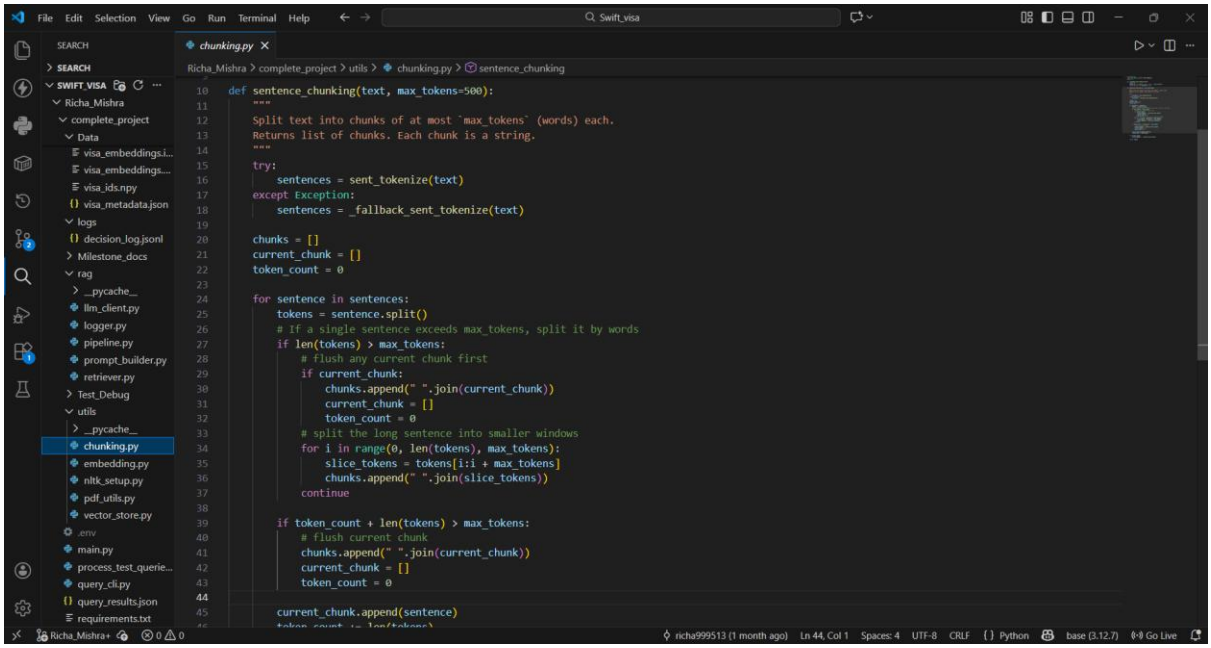
- **Role:** Splits large visa documents into smaller, manageable text chunks.
- **Importance:** Chunking ensures that embeddings capture localized context and improves retrieval accuracy.
- **Key Functions:**
    - Sentence/paragraph segmentation.
    - Fixed-size chunk creation (e.g., 500 tokens).

👉 ***Pseudocode/Code Snippet:***

MODULE chunking

FUNCTION _fallback_sent_tokenize(text): RETURN sentences

FUNCTION sentence_chunking(text, max=500): RETURN chunks



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***:

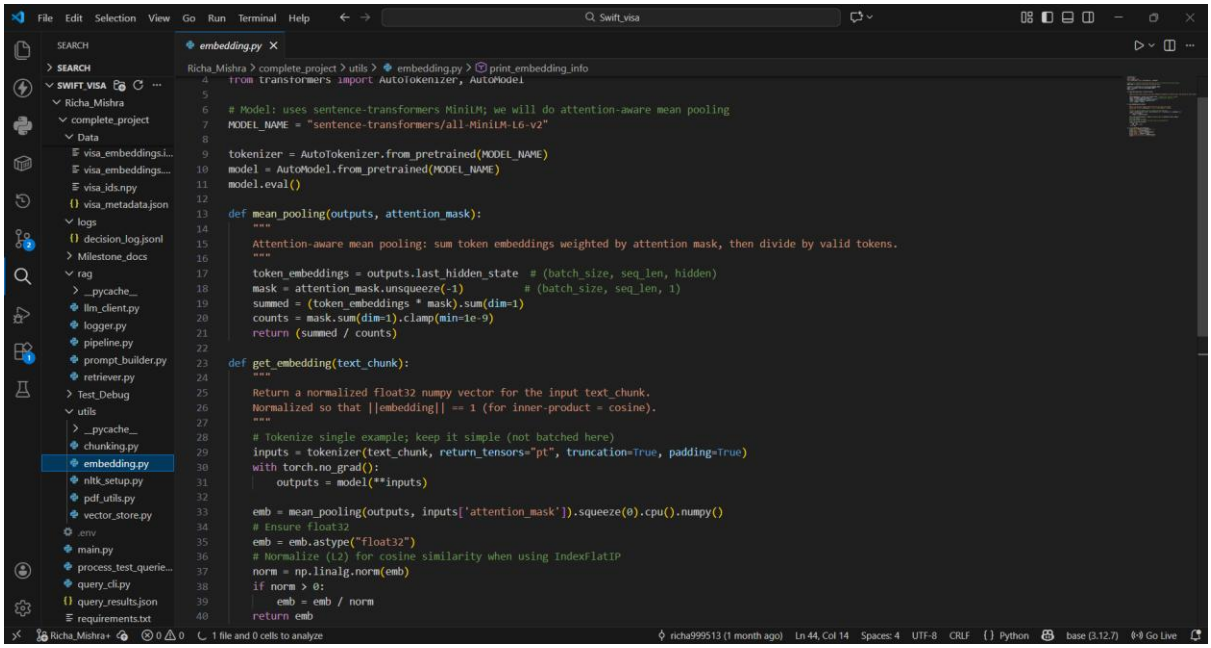
MODULE embedding

FUNCTION mean_pooling(out, mask): RETURN pooled_vector

FUNCTION get_embedding(chunk): RETURN normalized_vector

FUNCTION print_embedding_info(vec): PRINT diagnostics

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.

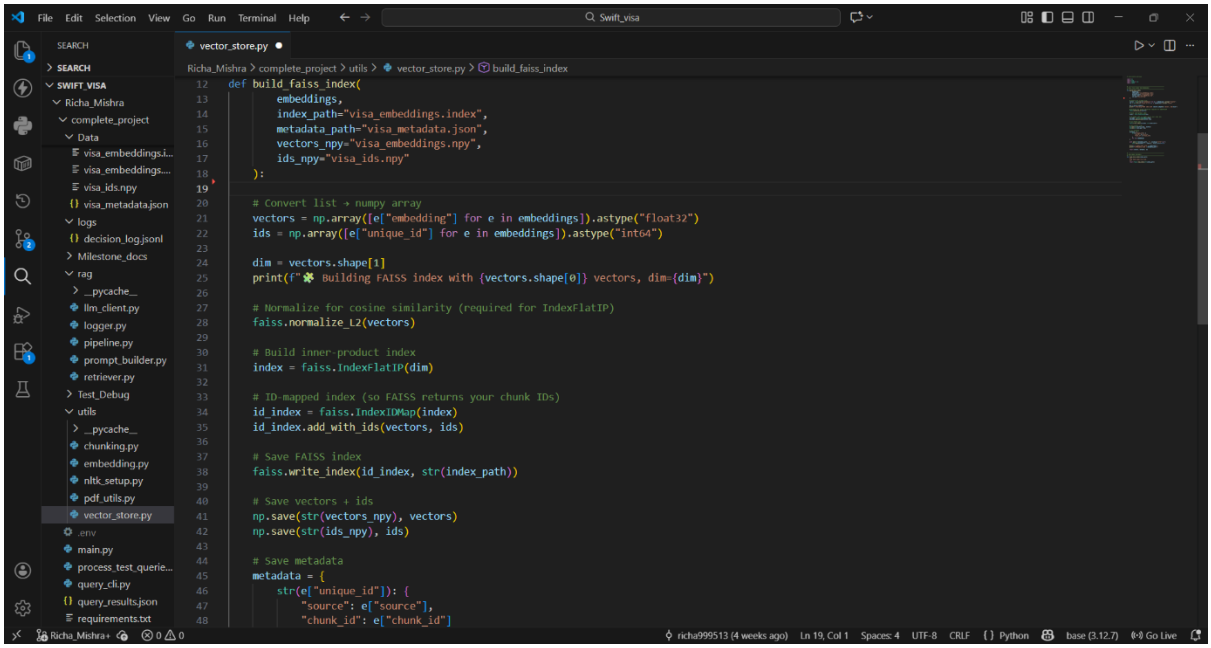
👉 ***Pseudocode/Code Snippet:***

MODULE vector_store

FUNCTION build_faiss_index(emb, ip, mp, vp, idp): RETURN index_files

FUNCTION load_faiss_index(path): RETURN 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

CALL ensure_nltk_resources()

FUNCTION process_documents():

FOR each file in Data/pdfs:

IF pdf → text = extract_text_from_pdf(file)

ELSE IF txt → text = read_text_file(file)

IF text empty → SKIP

chunks = sentence_chunking(text)

FOR each chunk:

vec = get_embedding(chunk)

STORE {uid, source, vec}

uid++

SAVE chunks JSON

RETURN embeddings

END FUNCTION

MAIN:

emb = process_documents()

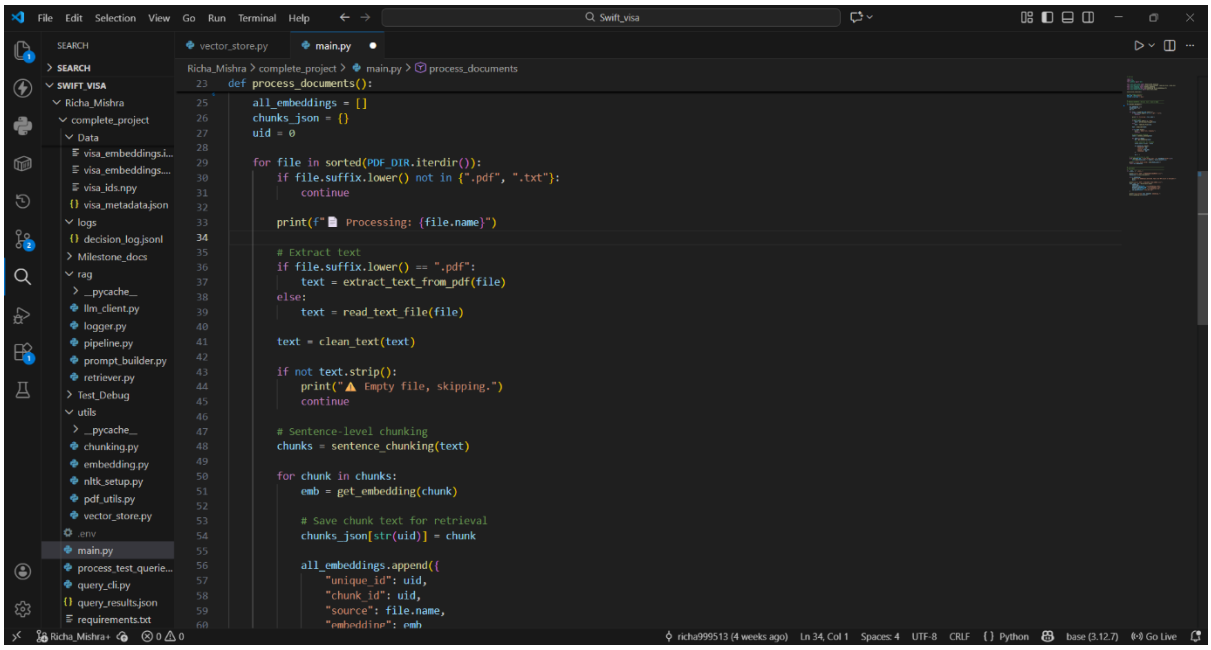
IF emb empty → EXIT

CALL build_faiss_index(emb, save_paths)

PRINT stored_count

CALL print_embedding_info(first_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.

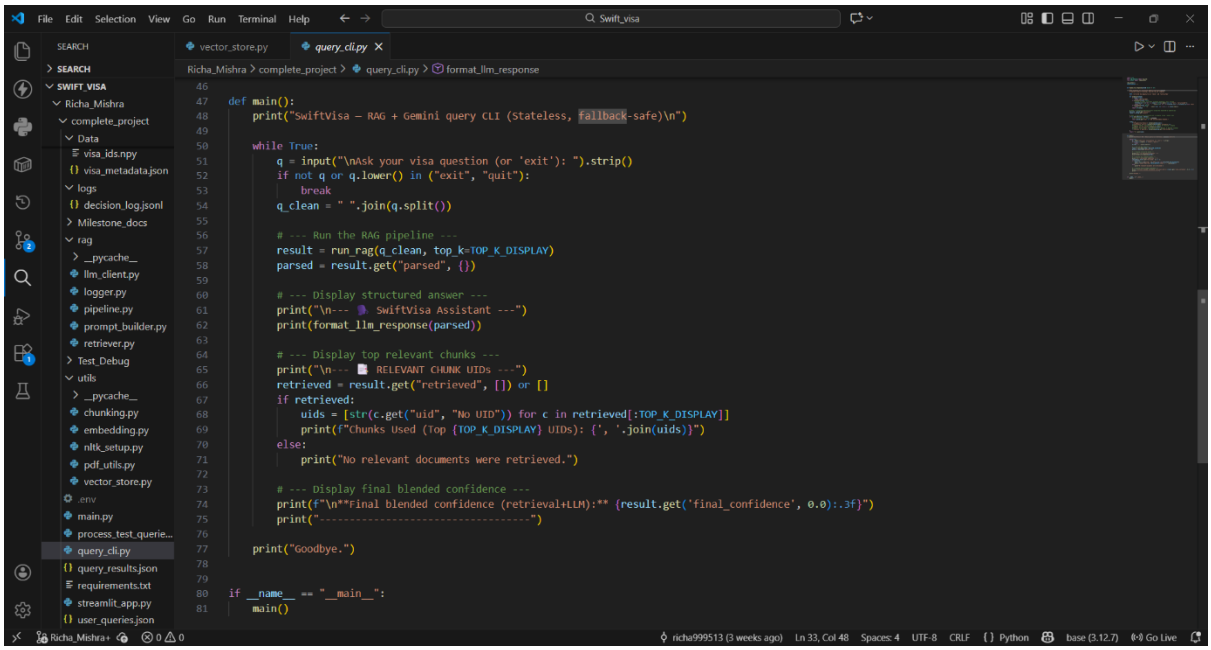
👉 ***Pseudocode/Code Snippet***:

FUNCTION format_llm_response(data): RETURN formatted_output

FUNCTION main():

  LOOP query → rag() → PRINT answer + chunk_uids + confidence

END FUNCTION



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

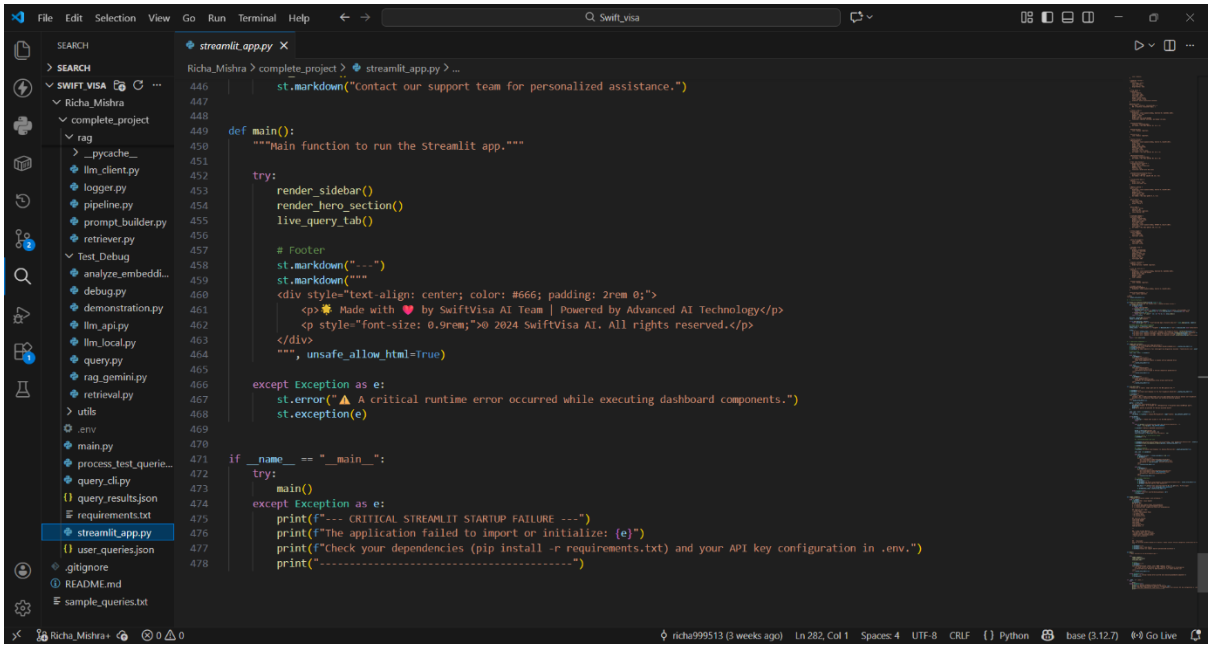
Show footer with credits

ENTRY:

IF run as script:

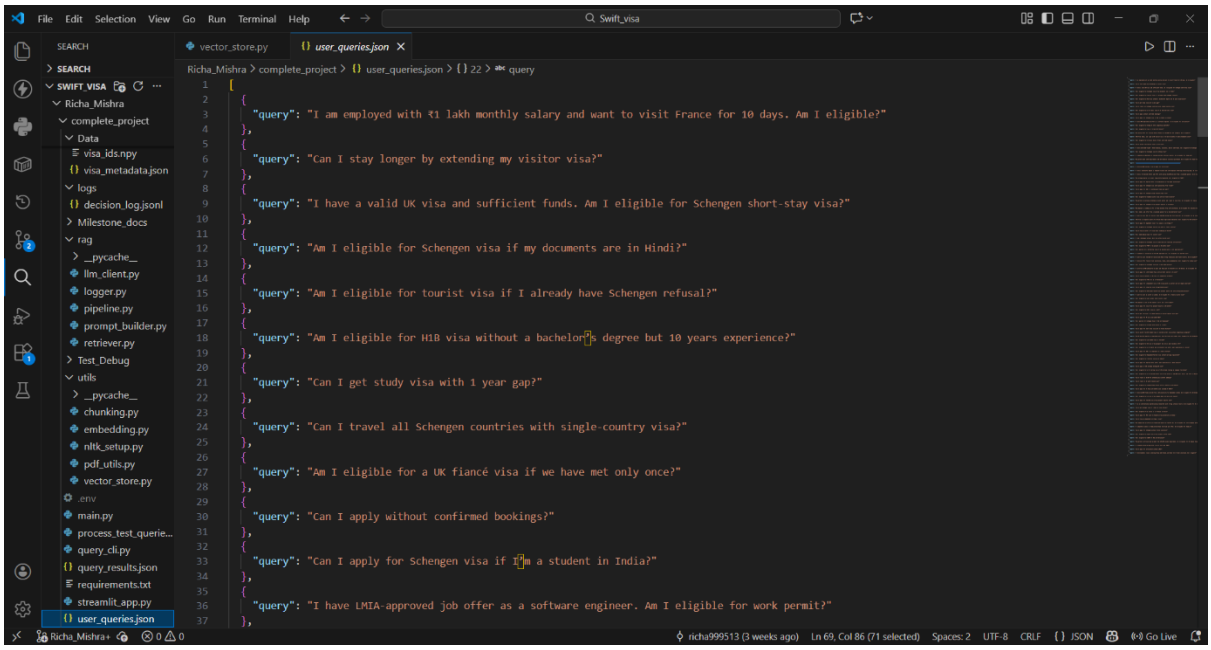
TRY run main()

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.

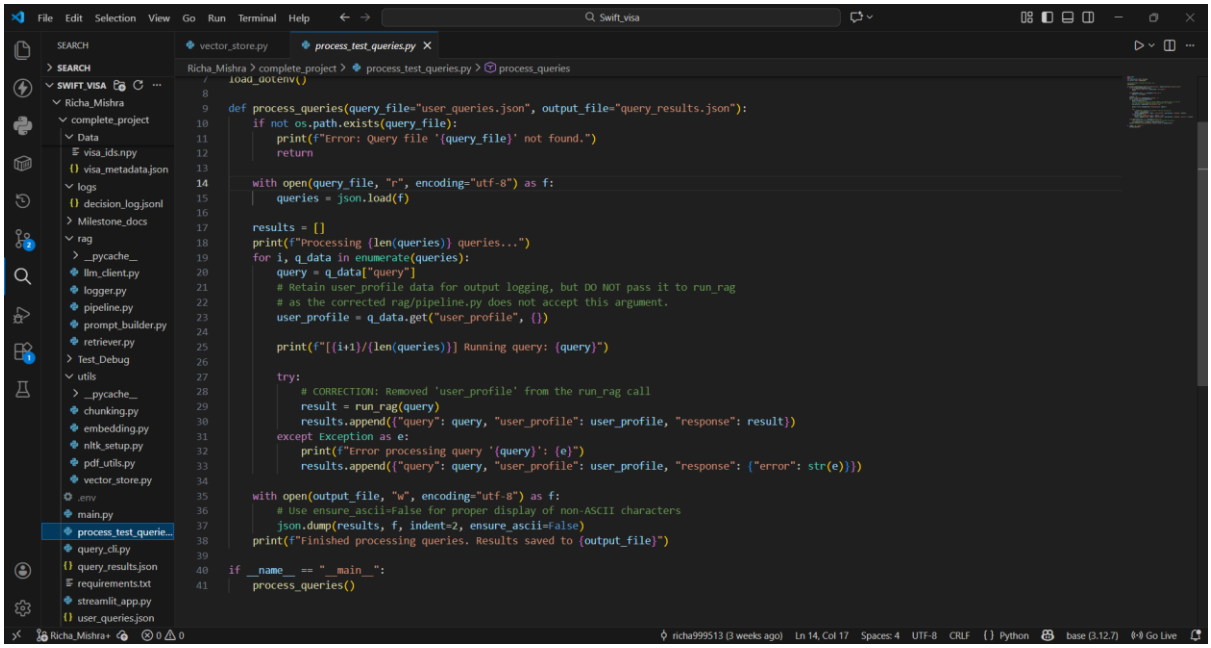


d. *process_test_queries.py*

- **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.

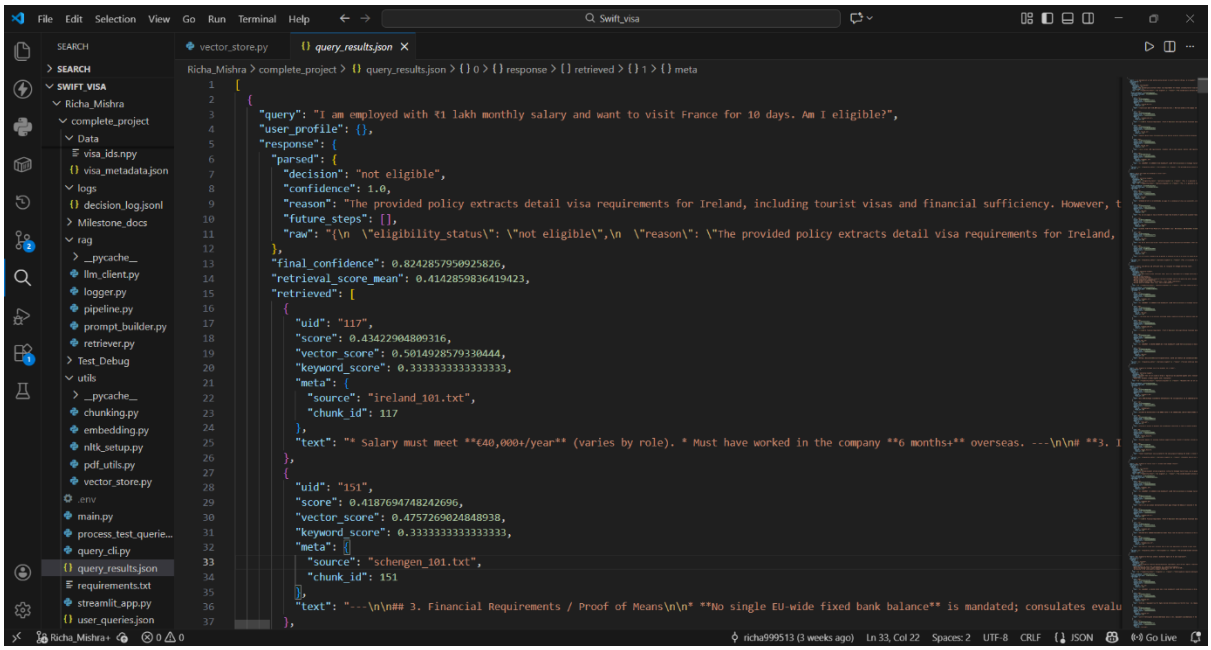
👉 ***Pseudocode/Code Snippet:***

```
FUNCTION process_queries(file, out):
    FOR q in queries: TRY rag(q) EXCEPT log_error
    SAVE results
END FUNCTION
```



```python
load_dotenv()

def process_queries(query_file="user_queries.json", output_file="query_results.json"):
    if not os.path.exists(query_file):
        print(f"Error: Query file '{query_file}' not found.")
        return

    with open(query_file, "r", encoding="utf-8") as f:
        queries = json.load(f)

    results = []
    print(f"Processing {len(queries)} queries...")
    for i, q_data in enumerate(queries):
        query = q_data["query"]
        # Retain user_profile data for output logging, but DO NOT pass it to run_rag
        # as the corrected rag/pipeline.py does not accept this argument.
        user_profile = q_data.get("user_profile", {})

        print(f"[{i+1}/{len(queries)}] Running query: {query}")

        try:
            # CORRECTION: Removed 'user_profile' from the run_rag call
            result = run_rag(query)
            results.append({"query": query, "user_profile": user_profile, "response": result})
        except Exception as e:
            print(f"Error processing query '{query}': {e}")
            results.append({"query": query, "user_profile": user_profile, "response": {"error": str(e)}})

    with open(output_file, "w", encoding="utf-8") as f:
        # Use ensure_ascii=False for proper display of non-ASCII characters
        json.dump(results, f, indent=2, ensure_ascii=False)
    print(f"Finished processing queries. Results saved to {output_file}")

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

Query_results.json




```
[
  {
    "query": "I am employed with ₹1 lakh monthly salary and want to visit France for 10 days. Am I eligible?",
    "user_profile": {},
    "response": {
      "parsed": {
        "decision": "not eligible",
        "confidence": 1.0,
        "reason": "The provided policy extracts detail visa requirements for Ireland, including tourist visas and financial sufficiency. However, t
        "future_steps": [],
        "raw": "{\n  \"eligibility_status\": \"not eligible\",\n  \"reason\": \"The provided policy extracts detail visa requirements for Ireland,
      },
      "final_confidence": 0.8242857950925826,
      "retrieval_score_mean": 0.4142859836419423,
      "retrieved": [
        {
          "uid": "117",
          "score": 0.43422904809316,
          "vector_score": 0.5014928579330444,
          "keyword_score": 0.3333333333333333,
          "meta": {
            "source": "ireland_101.txt",
            "chunk_id": 117
          },
          "text": "* Salary must meet **€40,000+/year** (varies by role). * Must have worked in the company **6 months+** overseas. ---\n\n# **3. I
        },
        {
          "uid": "151",
          "score": 0.4187694748242696,
          "vector_score": 0.4757269024848938,
          "keyword_score": 0.3333333333333333,
          "meta": {
            "source": "schengen_101.txt",
            "chunk_id": 151
          },
          "text": "---\n\n## 3. Financial Requirements / Proof of Means\n\n* **No single EU-wide fixed bank balance** is mandated; consulates evalu
        },
```


2. Workflow

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.

---

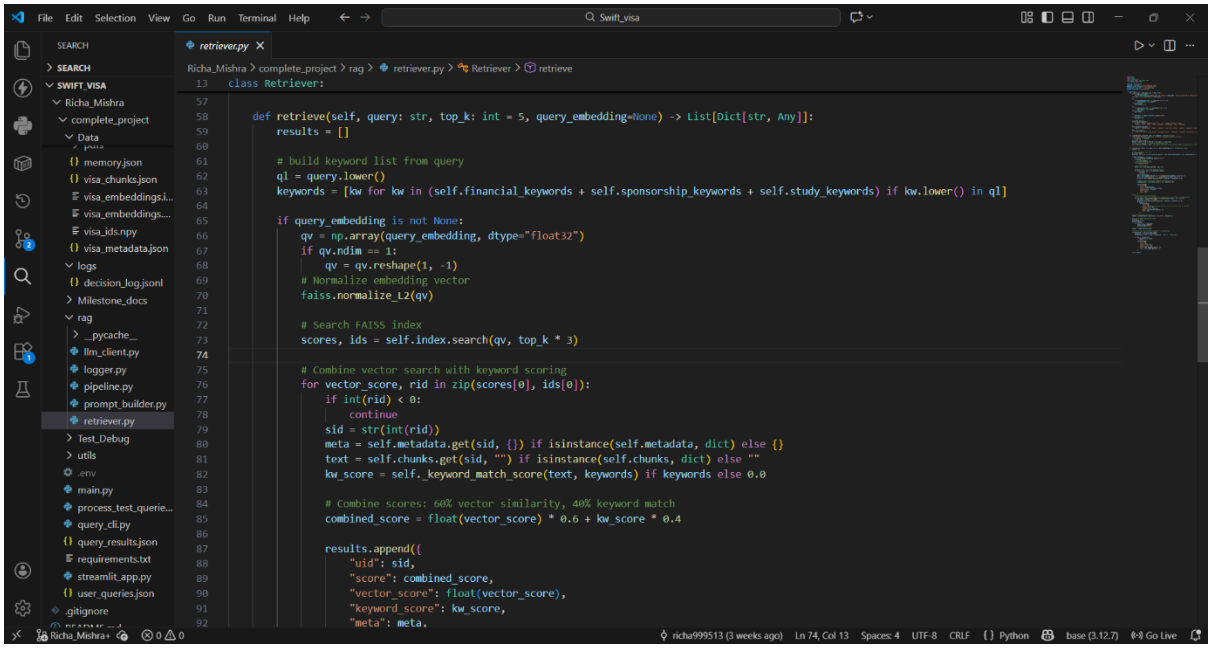
1. <u>Core Retrieval Components (rag/)</u>

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>***

```
CLASS Retriever

FUNCTION _keyword_match_score(t, k): RETURN score≤1

FUNCTION retrieve(q, k, emb):

   IF emb: SEARCH faiss → COMBINE vector+keyword → SORT → UNIQUE → TOP_K

   ELSE: KEYWORD_SCAN chunks

   RETURN results

END FUNCTION

END CLASS
```



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:***

FUNCTION get_embedding(q): RETURN vector_or_zero

FUNCTION compute_confidence_from_scores(s): RETURN mean_score

FUNCTION extract_info(txt): RETURN json_or_keyword_decision

FUNCTION run_rag(q, k):

v = embed(q)

r = retrieve(q,k,v)

p = build_prompt(q,r)

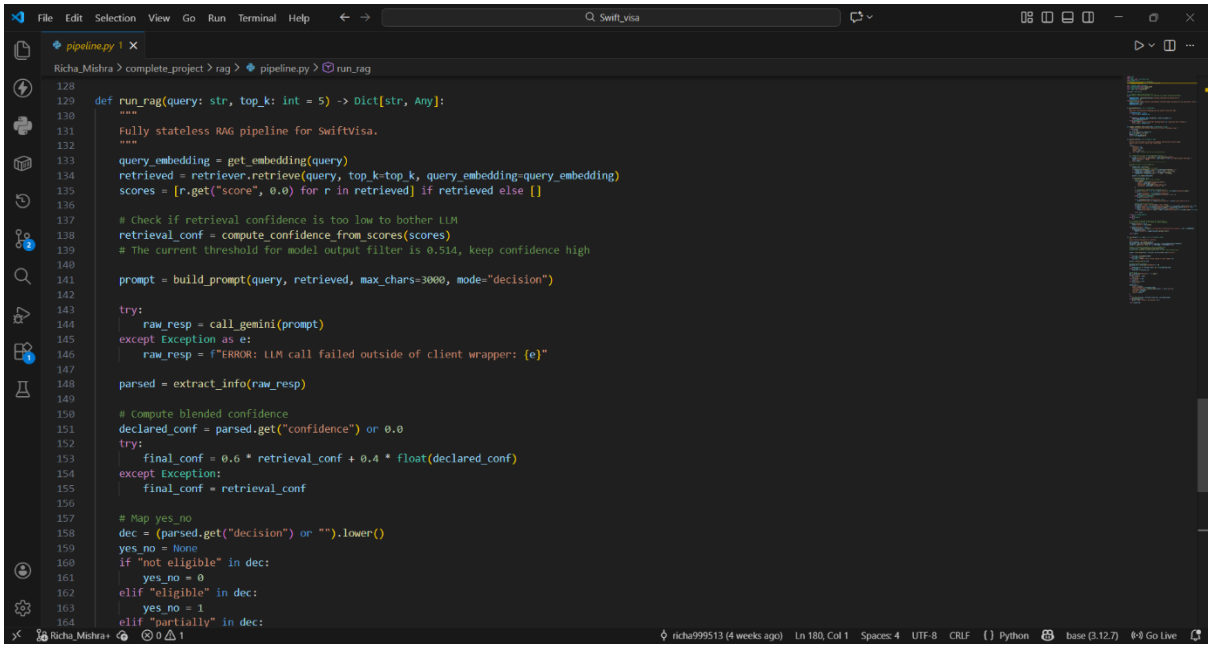
t = call_llm(p)

j = parse(t)

log_decision(q,r,j,p)

RETURN {j, confidence}

END FUNCTION



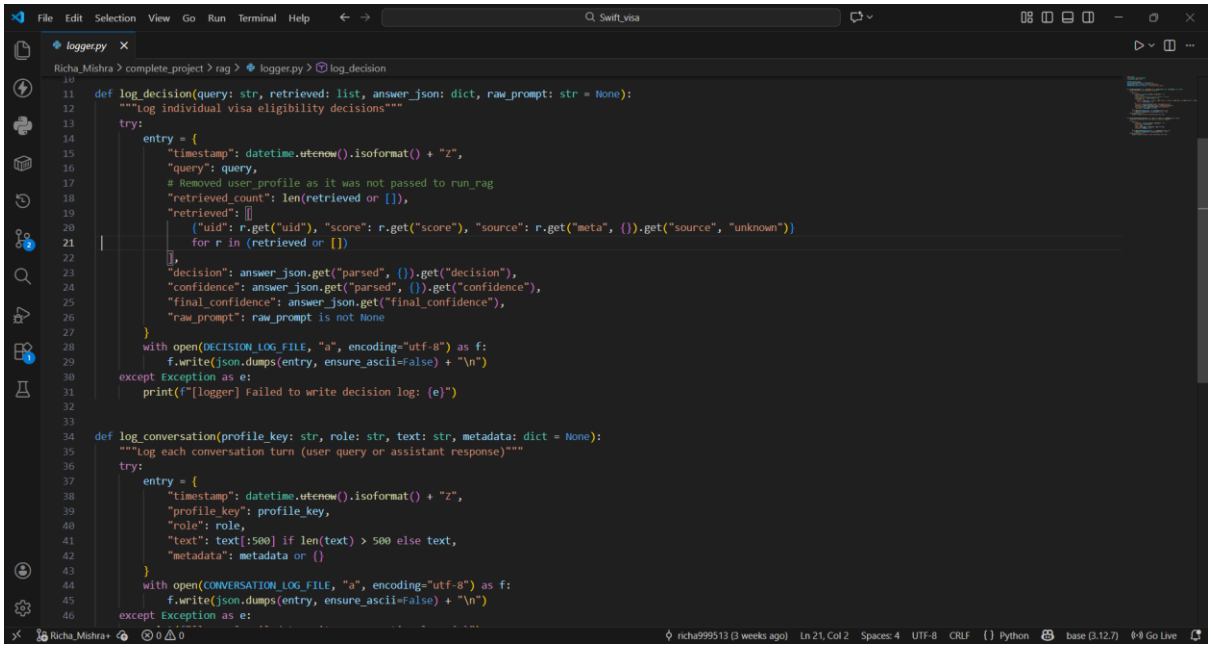
c. *logger.py*

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

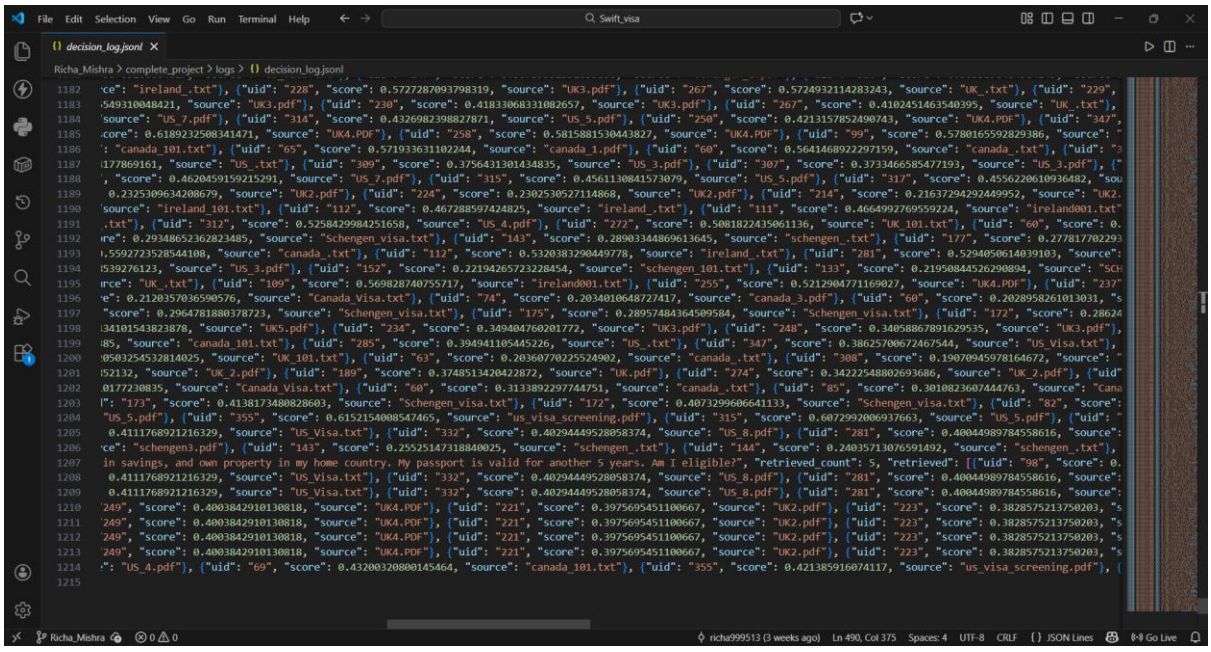
👉 ***Pseudocode/Code Snippet:***

FUNCTION log_decision(q, r, a, p?): SAVE json_line

FUNCTION log_conversation(pk, role, txt, meta?): SAVE json_line



```python
def log_decision(query: str, retrieved: list, answer_json: dict, raw_prompt: str = None):
    """Log individual visa eligibility decisions"""
    try:
        entry = {
            "timestamp": datetime.utcnow().isoformat() + "Z",
            "query": query,
            # Removed user_profile as it was not passed to run_rag
            "retrieved_count": len(retrieved or []),
            "retrieved": [
                {"uid": r.get("uid"), "score": r.get("score"), "source": r.get("meta", {}).get("source", "unknown")}
                for r in (retrieved or [])
            ],
            "decision": answer_json.get("parsed", {}).get("decision"),
            "confidence": answer_json.get("parsed", {}).get("confidence"),
            "final_confidence": answer_json.get("final_confidence"),
            "raw_prompt": raw_prompt is not None
        }
        with open(DECISION_LOG_FILE, "a", encoding="utf-8") as f:
            f.write(json.dumps(entry, ensure_ascii=False) + "\n")
    except Exception as e:
        print(f"[logger] Failed to write decision log: {e}")


def log_conversation(profile_key: str, role: str, text: str, metadata: dict = None):
    """Log each conversation turn (user query or assistant response)"""
    try:
        entry = {
            "timestamp": datetime.utcnow().isoformat() + "Z",
            "profile_key": profile_key,
            "role": role,
            "text": text[:500] if len(text) > 500 else text,
            "metadata": metadata or {}
        }
        with open(CONVERSATION_LOG_FILE, "a", encoding="utf-8") as f:
            f.write(json.dumps(entry, ensure_ascii=False) + "\n")
    except Exception as e:
```



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

a. ***prompt_builder.py***

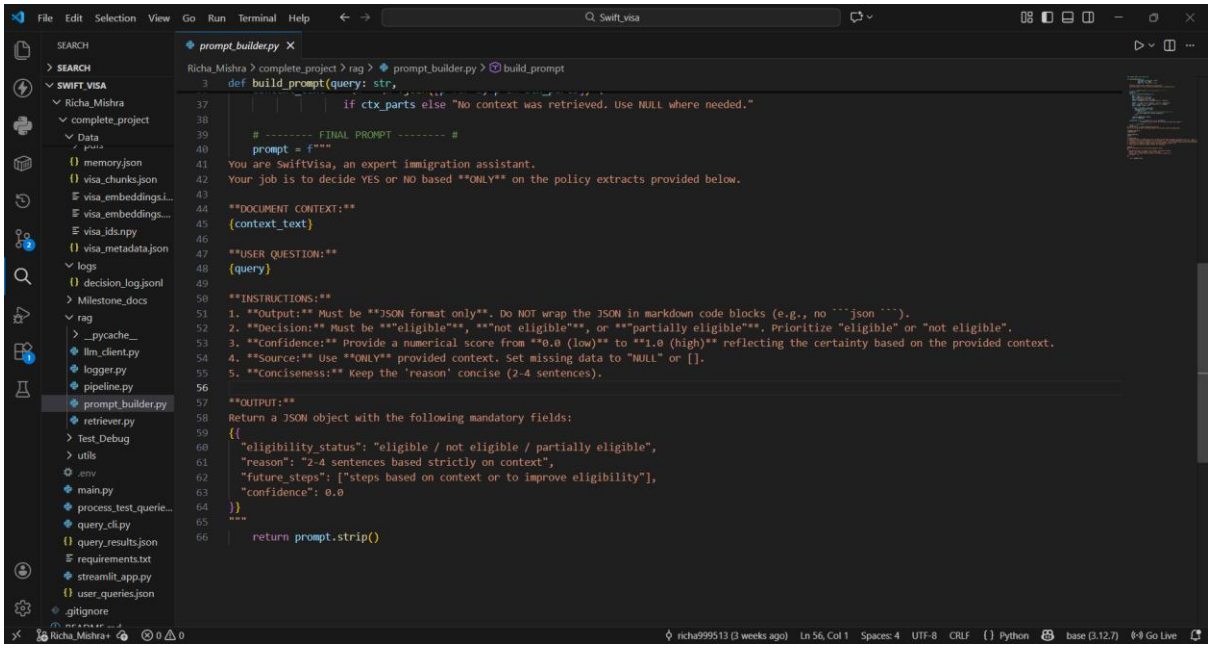
- **Role:** Builds structured prompts by combining the user query with retrieved chunks.

  Key Functions:

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

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

FUNCTION build_prompt(q, r, max, mode): RETURN final_prompt

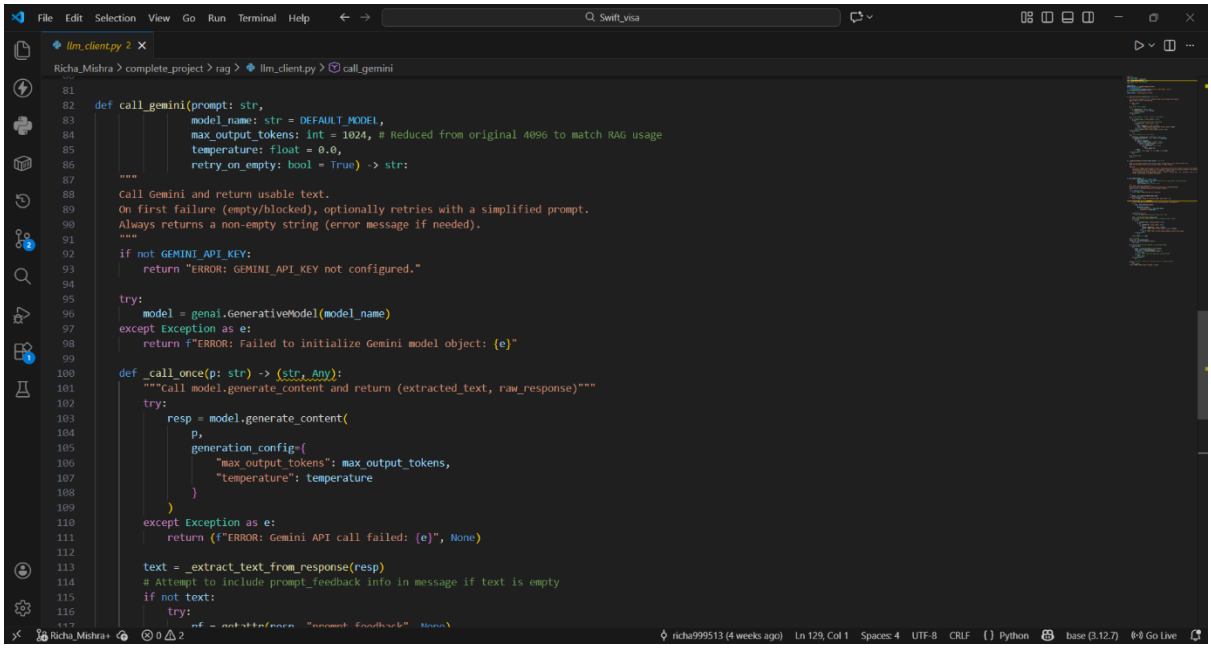


b. *llm_client.py*

- **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***:

FUNCTION call_gemini(prompt, model, tokens, temp, retry?): RETURN text_or_error



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.

2. Workflow

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. Output Components

a. *Response Display*

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

👉 ***Pseudocode/Code Snippet***:

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.

👉 ***Pseudocode/Code Snippet***:

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]

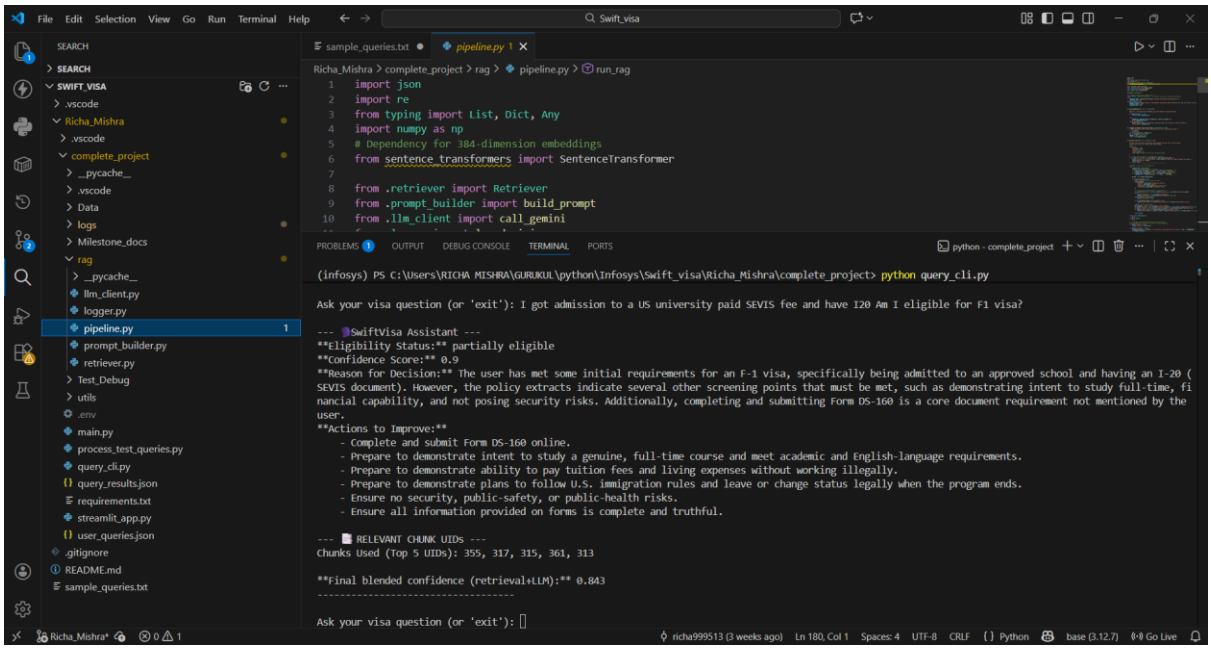
Top 5 Related Chunks: [IDs]

Eligibility Reasoning: [Explanation]

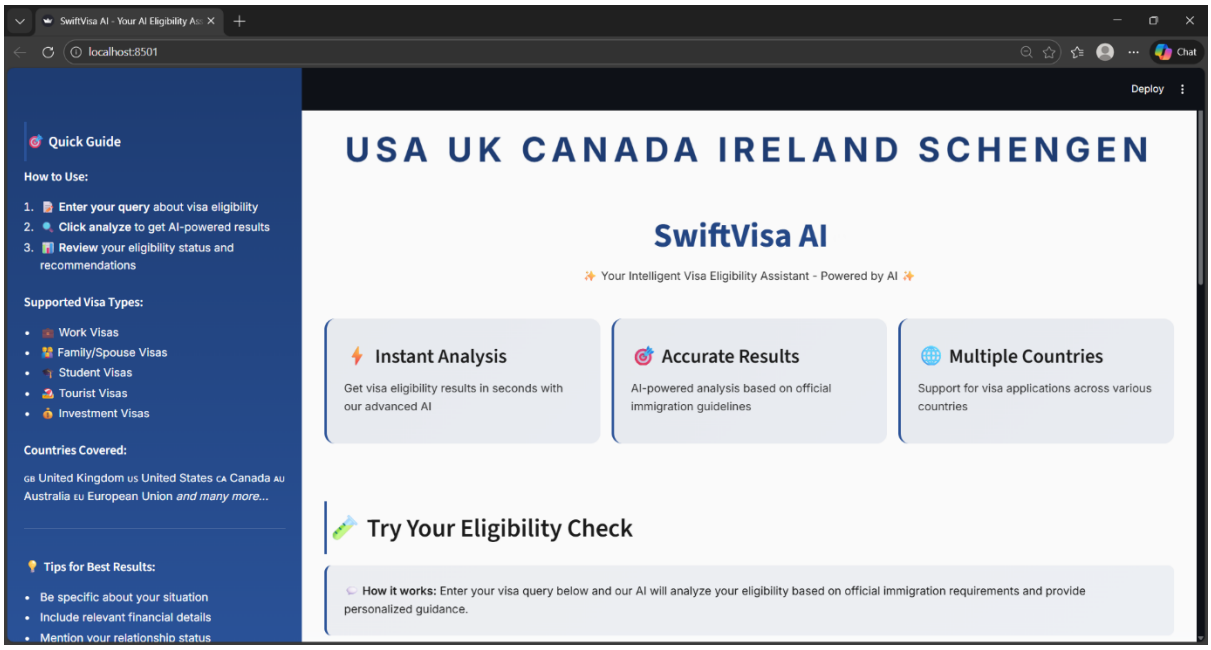
Future Steps: [Suggested actions]

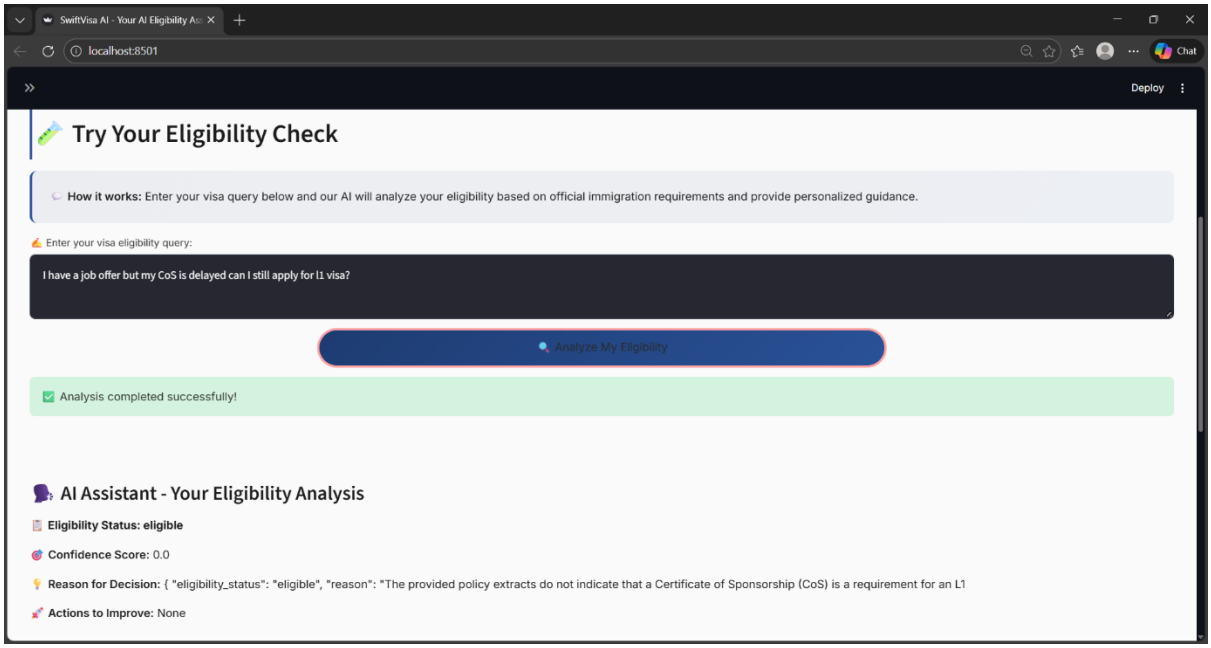
Output Screen:

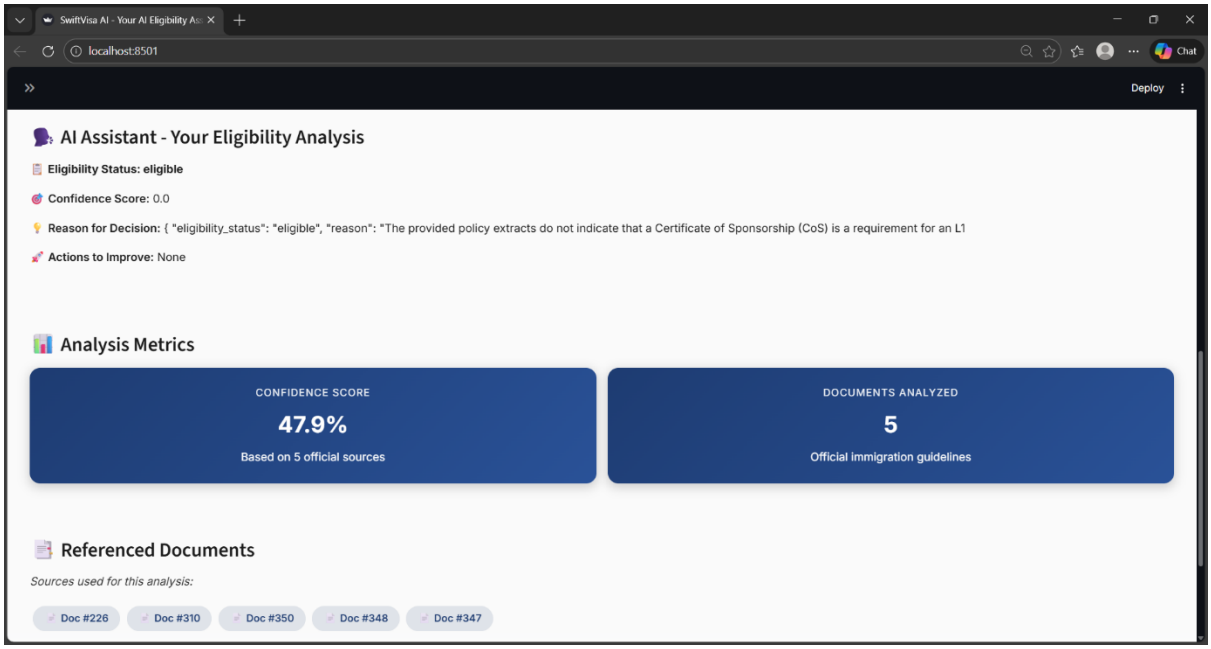
CLI Chatbot Screen:



Streamlit Dashboard Screen:







Answers to Some Important Questions:

Why Vector Database is used (Point-wise & Precise)

- Enables semantic similarity search (understands meaning, not just exact matches).
- Retrieves relevant visa guidelines even if queries use different wording/synonyms.
- Uses efficient vector indexing algorithms (HNSW, IVF, PQ) for fast nearest-neighbor search.
- More scalable than Python lists for millions of embeddings.
- Supports metadata filtering + hybrid search (vector + keyword), unlike Python lists.
- Provides persistent storage, real-time retrieval, and distributed search, which lists cannot.
- Ideal for AI agents in Swift Visa AI to deliver accurate document and rule recommendations quickly.

Popular Vector Databases + When to Use Them

Vector DB	Best Used When
Pinecone	Need managed, cloud-native, fast real-time search for production (good for Swift Visa AI deployment)
Milvus / Zilliz Cloud	Large-scale vector search with advanced indexing and open-source flexibility
Weaviate	Want built-in hybrid search + metadata filters + easy REST/GraphQL integration
Qdrant	Need lightweight, fast, open-source DB with payload filtering
ChromaDB	Prototyping, local storage, quick testing of embeddings
FAISS	Only vector search needed inside Python app, no DB features like metadata or user access
MongoDB Atlas Vector Search	Already using MongoDB and want integrated vector + traditional data storage

Embedding Transformer Models for Text → Vector Conversion

You can use these embedding transformers depending on accuracy, speed, and domain need:

Provider	Embedding Model Name
OpenAI	text-embedding-3-large, text-embedding-3-small
Hugging Face (Sentence Transformers)	all-mpnet-base-v2, all-MiniLM-L6-v2, multi-qa-mpnet-base-dot-v1
Google	embedding-001, text-embedding-004
Cohere	embed-english-v3.0, embed-multilingual-v3.0

Provider	Embedding Model Name
Meta	LLaMA-based embeddings (via HF), E5 embeddings
Microsoft	text-embedding-ada-002 (older but usable), E5-large-v2, E5-small-v2

Different Similarity Search Approaches

1. Exact Similarity Search

- Brute force / Linear scan
- Compares query vector with all stored vectors
- 100% accurate but slow for large data

2. Approximate Nearest Neighbor (ANN) Search

- Trades slight accuracy for very fast results
- Uses indexing algorithms like:
    - HNSW (graph-based)
    - IVF (cluster-based)
    - PQ (compressed vector search)

3. Similarity by Distance Metrics

You can compute similarity using mathematical metrics:

Method	How it works
Cosine Similarity	Measures angle between vectors (best for semantic text search like in Swift Visa AI)
Euclidean Distance (L2)	Measures straight-line distance
Dot Product Similarity	Higher score = more similar

4. Hybrid Similarity Search

- Combines keyword search + vector semantic search
- Useful when exact terms (e.g., country name, visa type) matter along with meaning

5. Re-Ranking after Similarity Search

- Retrieve top N results using ANN
- Then re-score and reorder using exact similarity or an LLM
- Boosts final accuracy

Different LLMs

LLM / Model	Suitable Use Case in Swift Visa AI	Free or Paid
GPT-4o / GPT-5 series (OpenAI API)	Advanced reasoning, document guidance, policy answering, structured report generation	Paid API
Claude 3.5 / Claude 4 (Anthropic API)	Long context understanding, detailed visa rule explanation, strong multilingual assistance	Paid API
Cohere Command R+	Retrieval-augmented answers, enterprise-level recommendations, multilingual QA	Paid
LLaMA 3 / 3.1 / 3.2 (Meta)	Local/self-hosted semantic QA, cost-efficient AI agent backbone	Free (open-source)
Mistral 7B / Mixtral (Mistral AI)	Local embeddings + QA, fast structured responses, self-hosting	Free locally, Paid if cloud-hosted
DeepSeek-R1 / DeepSeek-V3	Structured visa answers, form-like response generation, reasoning-heavy queries	Free (open models)
Qwen 2.5 / Qwen Turbo (Alibaba)	Strong multilingual intent matching and precise visa guidance	Free (open-source)
Gemma 2 (Google)	Local lightweight reasoning, policy summary, semantic QA	Free (open-source)
Gemini 1.5/2.0 Flash (Google API)	Fast visa answers, prototype API use, multilingual QA	Free tier available, Paid at scale