



INTERNSHIP PROJECT REPORT

SwiftVisaAI – RAG Based Visa Eligibility Screening Agent

Submitted by:

Richa Mishra

Internship Role: **AI/ML Intern**

Project Duration: **November 2025 – January 2026**

Submitted to:

Infosys

Mentored by:

Mr. Siddarth

Position: **Data Scientist**

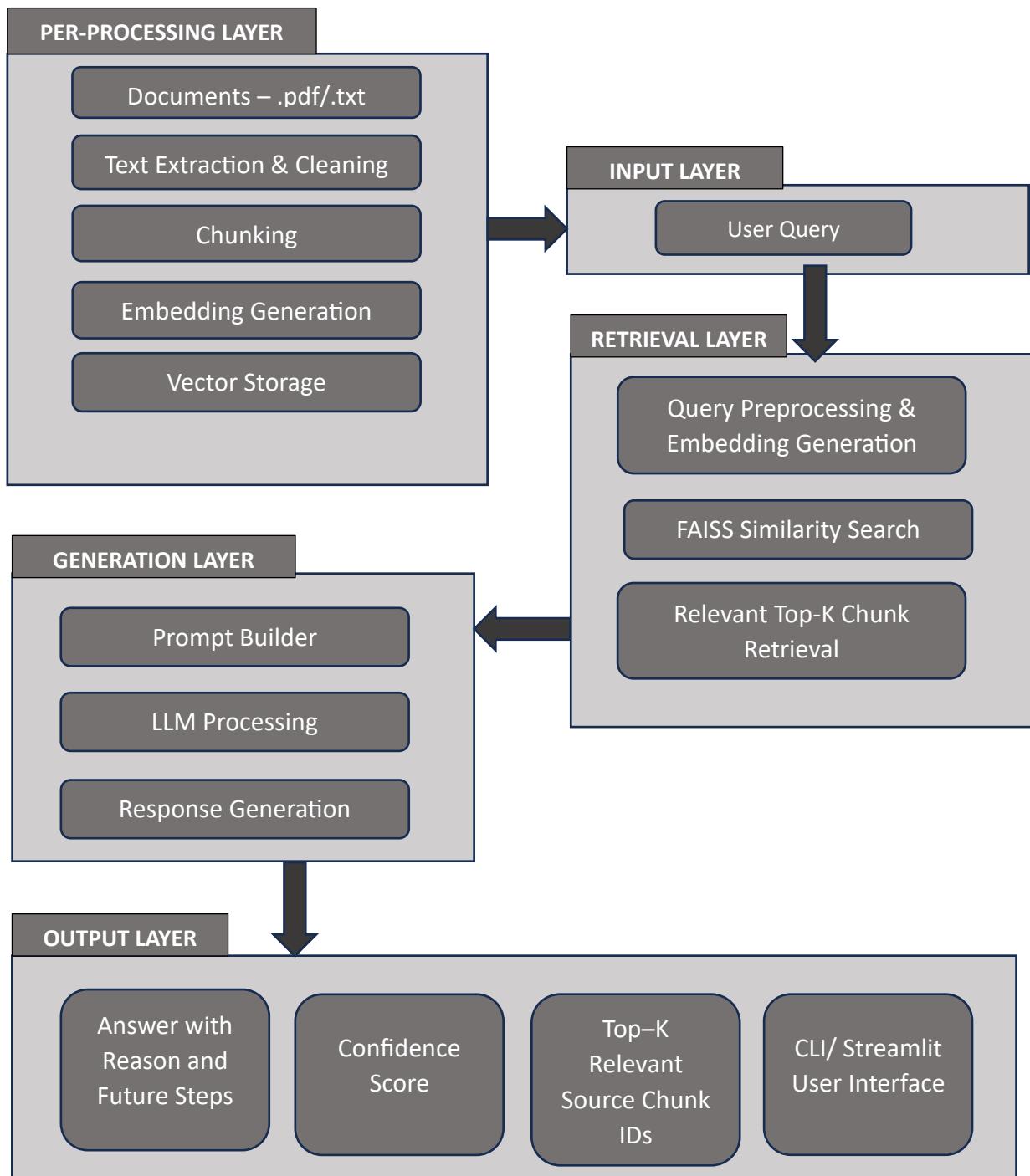
Submission Date:

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

The screenshot shows a code editor interface with the following details:

- Project Structure:** The left sidebar shows a tree view of the project structure under "Richa_Mishra > complete_project > utils". The "pdf_utils.py" file is selected.
- Code Editor:** The main pane displays the content of the "pdf_utils.py" file. The code defines a function "extract_text_from_pdf" that uses pdfplumber to extract text from PDFs. If text is missing, it falls back to OCR using Pytesseract. The code includes comments explaining the fallback logic and handling of empty results.
- Status Bar:** The bottom status bar shows "Line 112, Col 1" and other standard status indicators.

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

```
Richa_Mishra > complete_project > utils > chunking.py > sentence_chunking
10 def sentence_chunking(text, max_tokens=500):
11     """
12         Split text into chunks of at most `max_tokens` (words) each.
13         Returns list of chunks. Each chunk is a string.
14     """
15     try:
16         sentences = sent_tokenize(text)
17     except Exception:
18         sentences = _fallback_sent_tokenize(text)
19
20     chunks = []
21     current_chunk = []
22     token_count = 0
23
24     for sentence in sentences:
25         tokens = sentence.split()
26         if len(tokens) > max_tokens:
27             # if a single sentence exceeds max_tokens, split it by words
28             if len(current_chunk) > 0:
29                 # flush any current chunk first
30                 if current_chunk:
31                     chunks.append(" ".join(current_chunk))
32                     current_chunk = []
33                     token_count = 0
34             # split the long sentence into smaller windows
35             for i in range(0, len(tokens), max_tokens):
36                 slice_tokens = tokens[i:i + max_tokens]
37                 chunks.append(" ".join(slice_tokens))
38             continue
39
40         if token_count + len(tokens) > max_tokens:
41             # flush current chunk
42             chunks.append(" ".join(current_chunk))
43             current_chunk = []
44             token_count = 0
45
46         current_chunk.append(sentence)
47         token_count += len(tokens)
```

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

```
Richa_Mishra > complete.project > utils > embedding.py > print_embedding_info
4     from transformers import AutoTokenizer, AutoModel
5
6     # Model: uses sentence-transformers MiniLM; we will do attention-aware mean pooling
7     MODEL_NAME = "sentence-transformers/all-MiniLM-L6-v2"
8
9     tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
10    model = AutoModel.from_pretrained(MODEL_NAME)
11    model.eval()
12
13    def mean_pooling(outputs, attention_mask):
14        """
15            Attention-aware mean pooling: sum token embeddings weighted by attention mask, then divide by valid tokens.
16        """
17        token_embeddings = outputs.last_hidden_state # (batch_size, seq_len, hidden)
18        mask = attention_mask.unsqueeze(-1)           # (batch_size, seq_len, 1)
19        summed = (token_embeddings * mask).sum(dim=1)
20        counts = mask.sum(dim=1).clamp(min=1e-9)
21        return (summed / counts)
22
23    def get_embedding(text_chunk):
24        """
25            Return a normalized float32 numpy vector for the input text_chunk.
26            Normalized so that ||embedding|| == 1 (for inner-product = cosine).
27        """
28        # Tokenize single example; keep it simple (not batched here)
29        inputs = tokenizer(text_chunk, return_tensors="pt", truncation=True, padding=True)
30        with torch.no_grad():
31            outputs = model(**inputs)
32
33        emb = mean_pooling(outputs, inputs['attention_mask']).squeeze(0).cpu().numpy()
34        # Ensure float32
35        emb = emb.astype("float32")
36        # Normalize (L2) for cosine similarity when using IndexFlatIP
37        norm = np.linalg.norm(emb)
38        if norm > 0:
39            emb = emb / norm
40
41        return emb
```

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
```

```

File Edit Selection View Go Run Terminal Help ← →
C: Swift_visa
SEARCH ⚡ SEARCH
SWIFT_VISA
Richa_Mishra
complete_project
Data
visa_embeddings...
visa_embeddings...
visa_ids.npy
visa_metadata.json
logs
decision_log.json
Milestone_docs
rag
_pycache_
llm_client.py
logger.py
pipeline.py
prompt_builder.py
retriever.py
Test_Debug
utils
_pycache_
chunking.py
embedding.py
nltk_setup.py
pdf_utils.py
vector_store.py
.env
main.py
process_test_querie...
query_cli.py
query_results.json
requirements.txt
vector_store.py
12 def build_faiss_index(
13     embeddings,
14     index_path="visa_embeddings.index",
15     metadata_path="visa_metadata.json",
16     vectors_npy="visa_embeddings.npy",
17     ids_npy="visa_ids.npy"
18 ):
19     # Convert list to numpy array
20     vectors = np.array([e["embedding"] for e in embeddings]).astype("float32")
21     ids = np.array([e["unique_id"] for e in embeddings]).astype("int64")
22
23     dim = vectors.shape[1]
24     print(f"Building FAISS index with {vectors.shape[0]} vectors, dim={dim}")
25
26     # Normalize for cosine similarity (required for IndexFlatIP)
27     faiss.normalize_L2(vectors)
28
29     # Build inner-product index
30     index = faiss.IndexFlatIP(dim)
31
32     # ID-mapped index (so FAISS returns your chunk IDs)
33     id_index = faiss.IndexIDMap(index)
34     id_index.add_with_ids(vectors, ids)
35
36     # Save FAISS index
37     faiss.write_index(id_index, str(index_path))
38
39     # Save vectors + ids
40     np.save(str(vectors_npy), vectors)
41     np.save(str(ids_npy), ids)
42
43     # Save metadata
44     metadata = [
45         str(e["unique_id]): {
46             "source": e["source"],
47             "chunk_id": e["chunk_id"]
48         }
49     ]
49
50     with open(metadata_path, "w") as f:
51         f.write(json.dumps(metadata))
52
53     print(f"Written FAISS index to {index_path} and vectors to {vectors_npy} and {ids_npy} and metadata to {metadata_path}")
54
55     return index
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
99

```

richa@99513 (4 weeks ago) Ln 19, Col 1 Spaces: 4 UTF-8 CRLF {} Python base (3.12.7) ⓘ Go Live

3. Workflow Integration ([main.py](#))

- Role:** Orchestrates the preprocessing pipeline by calling the utility scripts in sequence.
- Steps:**
 - Load raw visa documents from Data/.
 - Extract text using pdf_utils.py.
 - Chunk text with chunking.py.
 - Generate embeddings via embedding.py.
 - 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
```

The screenshot shows a code editor window titled "Swift_VISA". The left sidebar displays a file tree for a project named "complete_project" under "SWIFT_VISA". The main editor area contains the following Python code:

```
def process_documents():
    all_embeddings = []
    chunks_json = {}
    uid = 0

    for file in sorted(PDF_DIR.iterdir()):
        if file.suffix.lower() not in {".pdf", ".txt"}:
            continue

        print(f"Processing: {file.name}")

        # Extract text
        if file.suffix.lower() == ".pdf":
            text = extract_text_from_pdf(file)
        else:
            text = read_text_file(file)

        text = clean_text(text)

        if not text.strip():
            print("Empty file, skipping.")
            continue

        # Sentence-level chunking
        chunks = sentence_chunking(text)

        for chunk in chunks:
            emb = get_embedding(chunk)

            # Save chunk text for retrieval
            chunks_json[str(uid)] = chunk

            all_embeddings.append({
                "unique_id": uid,
                "chunk_id": uid,
                "source": file.name,
                "embedding": emb
            })

            uid += 1

    return all_embeddings, chunks_json
```

The status bar at the bottom indicates the file is "richa999513 (4 weeks ago)" with 1n 34, Col 1, Spaces: 4, UTF-8, CRLF, Python, base 3.12.7, Go Live.

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.

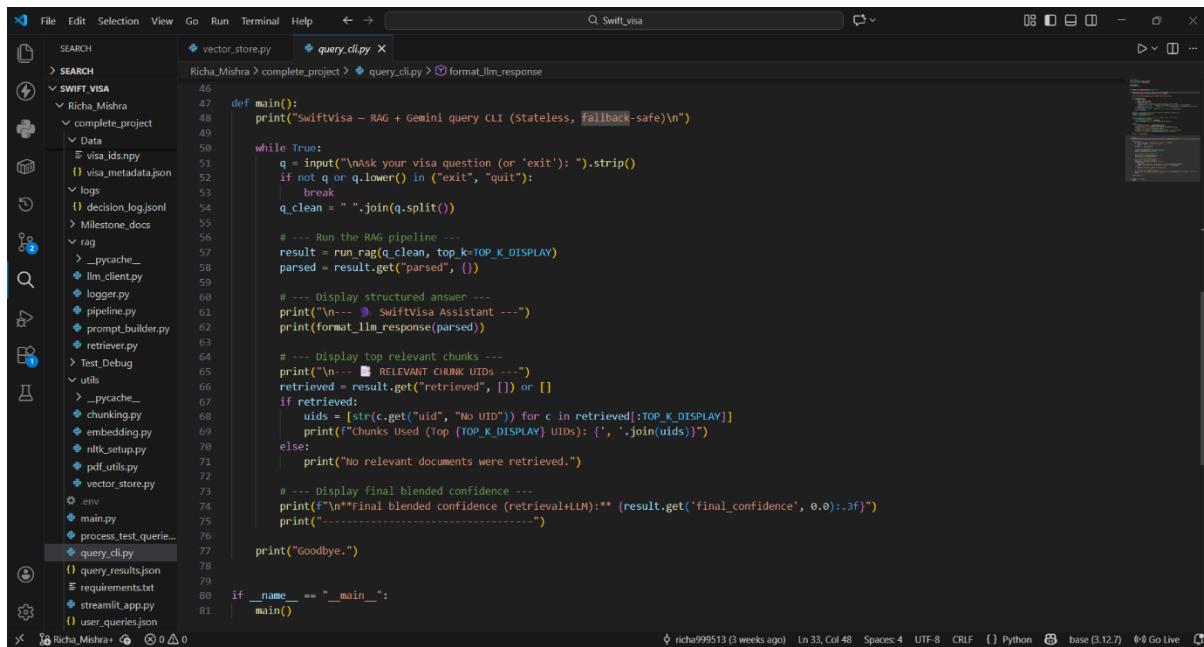
👉 Pseudocode/Code Snippet:

```
FUNCTION format_llm_response(data): RETURN formatted_output
```

```
FUNCTION main():
```

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

```
END FUNCTION
```



```
def format_llm_response(data):  
    RETURN formatted_output  
  
def main():  
    print("SwiftVisa - RAG + Gemini query CLI (stateless, fallback-safe)\n")  
    while True:  
        q = input("\nAsk your visa question (or 'exit'): ").strip()  
        if not q or q.lower() in ("exit", "quit"):  
            break  
        q_clean = " ".join(q.split())  
  
        # --- Run the RAG pipeline ---  
        result = run_rag(q_clean, top_k=TOP_K_DISPLAY)  
        parsed = result.get("parsed", {})  
  
        # --- Display structured answer ---  
        print("\n--- 🌐 SwiftVisa Assistant ---")  
        print(format_llm_response(parsed))  
  
        # --- Display top relevant chunks ---  
        print("\n--- 📄 RELEVANT CHUNK UIDS ---")  
        retrieved = result.get("retrieved", []) or []  
        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.")  
  
        # --- Display final blended confidence ---  
        print(f"\n**Final blended confidence (retrieval+LLM):** {result.get('final_confidence', 0.0):.3f}")  
        print("-----")  
  
        print("Goodbye.")  
    if __name__ == "__main__":  
        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

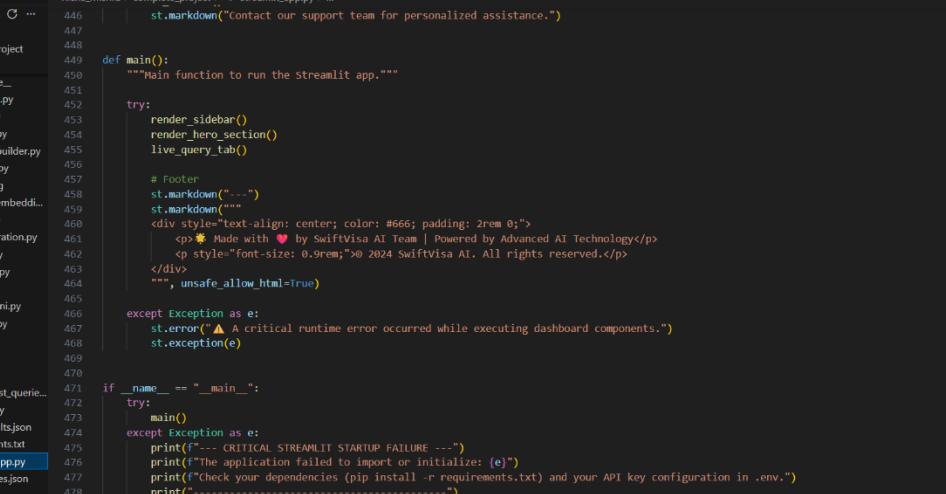
Show footer with credits

ENTRY:

IF run as script:

TRY run main()

CATCH startup errors → print critical failure message



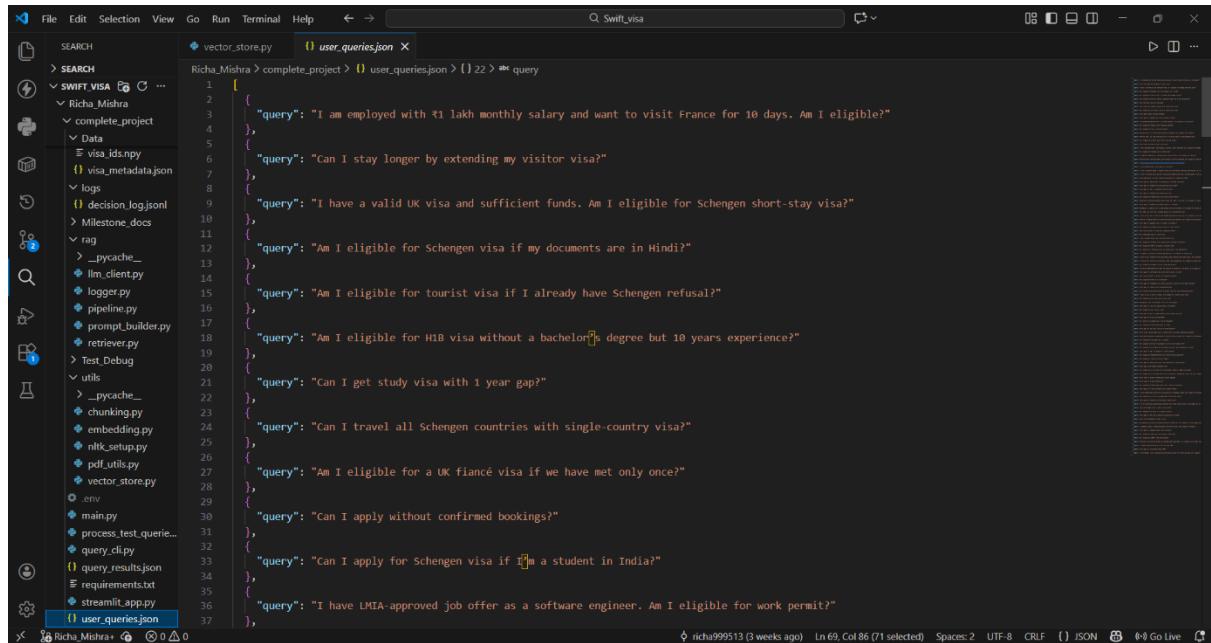
The screenshot shows a code editor interface with the following details:

- File Explorer:** On the left, it lists project files and folders:
 - SEARCH
 - > SEARCH
 - ✓ SWIFT VISA
 - Richa_Mishra
 - complete_project
 - > rag
 - _pycache_
 - ilm_client.py
 - logger.py
 - pipeline.py
 - prompt_builder.py
 - retriever.py
 - Test_Debug
 - analyze_embeddi...
 - debug.py
 - demonstration.py
 - ilm_api.py
 - ilm_local.py
 - query.py
 - rag_gemini.py
 - retrieval.py
 - > utils
 - env
 - main.py
 - process_test_querie...
 - query_clipy
 - query_results.json
 - requirements.txt
 - streamlit_app.py
 - user_queries.json
 - .gitignore
 - README.md
 - sample_queries.txt
- Search Bar:** At the top center, it says "Q Swift_visa".
- Code Editor:** The main area displays the `streamlit_app.py` file content. The code is a Streamlit application that runs a sidebar, displays a hero section, and handles a query tab. It includes a footer with AI credits and handles critical runtime errors. The file ends with a main block and a startup failure print statement.

```
streamlit_app.py x
Richa_Mishra > complete_project > streamlit_app.py > ...
446 |     st.markdown("Contact our support team for personalized assistance.")
447 |
448 |
449 def main():
450     """Main function to run the Streamlit app."""
451
452     try:
453         render_sidebar()
454         render_hero_section()
455         live_query_tab()
456
457         # Footer
458         st.markdown("---")
459         st.markdown("Made with ❤️ by SwiftVisa AI Team | Powered by Advanced AI Technology</p>")
460         st.markdown("© 2024 SwiftVisa AI. All rights reserved.</p>")
461     except Exception as e:
462         st.error('⚠️ A critical runtime error occurred while executing dashboard components.')
463         st.exception(e)
464
465     if __name__ == "__main__":
466         try:
467             main()
468         except Exception as e:
469             print("---- CRITICAL STREAMLIT STARTUP FAILURE ----")
470             print(f"the application failed to import or initialize: {e}")
471             print("Check your dependencies (pip install -r requirements.txt) and your API key configuration in .env.")
472             print("-----")
```

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.



```
vector.store.py user_queries.json
Richa_Mishra > complete_project > user_queries.json > 22 > * query
[{"query": "I am employed with ₹1 lakh monthly salary and want to visit France for 10 days. Am I eligible?", "answer": "Yes, you are eligible for a Schengen visa based on your employment and salary. However, you will need to provide proof of employment and sufficient funds for your trip."}, {"query": "Can I stay longer by extending my visitor visa?", "answer": "The duration of a visitor visa is typically limited to 90 days. If you require a longer stay, you may need to apply for a different type of visa, such as a residence permit or a work visa, depending on your circumstances."}, {"query": "I have a valid UK visa and sufficient funds. Am I eligible for Schengen short-stay visa?", "answer": "Yes, you are eligible for a Schengen short-stay visa if you have a valid UK visa and sufficient funds to support your stay in the Schengen area. You will also need to provide proof of accommodation and travel plans."}, {"query": "Am I eligible for Schengen visa if my documents are in Hindi?", "answer": "The language of your documents does not affect your eligibility for a Schengen visa. As long as you can demonstrate that you meet the requirements, your visa application will be processed regardless of the language of your documentation."}, {"query": "Am I eligible for tourist visa if I already have Schengen refusal?", "answer": "If you have previously been refused a Schengen visa, it is possible that you may still be eligible for a tourist visa, but your application will be subject to a more thorough review. You may need to provide additional documentation to demonstrate that you have learned from your previous refusal and no longer pose a risk to the Schengen area."}, {"query": "Am I eligible for H1B visa without a bachelor's degree but 10 years experience?", "answer": "Yes, it is possible to qualify for an H1B visa even if you do not have a bachelor's degree, provided that you have at least 10 years of relevant work experience. Your employer must also demonstrate that they cannot find a U.S. citizen or permanent resident to perform the job."}, {"query": "Can I get study visa with 1 year gap?", "answer": "Yes, it is possible to apply for a study visa even if there is a gap in your education. You will need to provide documentation to explain the gap and demonstrate that you are still eligible for the study visa."}, {"query": "Can I travel all Schengen countries with single-country visa?", "answer": "No, you cannot travel all Schengen countries with a single-country visa. Each country has its own visa requirements and entry rules. You will need to obtain a visa for each country you plan to visit, unless you are eligible for a Schengen免签证 (Schengen免签证)."}, {"query": "Am I eligible for a UK fiancé visa if we have met only once?", "answer": "Yes, it is possible to qualify for a UK fiancé visa even if you have only met once. You will need to demonstrate that you have a genuine relationship and that you intend to marry within two years of the visa being issued."}, {"query": "Can I apply without confirmed bookings?", "answer": "Yes, you are not required to have confirmed bookings before applying for a visa. You will need to provide documentation to demonstrate that you have a place to stay and that you have the financial means to support your stay."}, {"query": "Can I apply for Schengen visa if I'm a student in India?", "answer": "Yes, it is possible to apply for a Schengen visa if you are a student in India. You will need to provide documentation to demonstrate that you are a student and that you have a valid student visa or residence permit in India."}, {"query": "I have LMTA-approved job offer as a software engineer. Am I eligible for work permit?", "answer": "Yes, it is possible to qualify for a work permit if you have a LMTA-approved job offer as a software engineer. Your employer must demonstrate that they cannot find a U.S. citizen or permanent resident to perform the job."}]
```

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
```

```

Richa_Mishra > complete_project > process_test_queries.py > process_queries
    / load_dotenv()
    8
    9     def process_queries(query_file="user_queries.json", output_file="query_results.json"):
    10        if not os.path.exists(query_file):
    11            print(f"Error: Query file '{query_file}' not found.")
    12            return
    13
    14        with open(query_file, "r", encoding="utf-8") as f:
    15            queries = json.load(f)
    16
    17            results = []
    18            print(f"Processing {len(queries)} queries...")
    19            for i, q_data in enumerate(queries):
    20                query = q_data["query"]
    21                # Retain user_profile data for output logging, but DO NOT pass it to run_rag
    22                # as the corrected rag/pipeline.py does not accept this argument.
    23                user_profile = q_data.get("user_profile", {})
    24
    25                print(f"[{i+1}/{len(queries)}] Running query: {query}")
    26
    27                try:
    28                    # CORRECTION: Removed 'user_profile' from the run_rag call
    29                    result = run_rag(query)
    30                    results.append({"query": query, "user_profile": user_profile, "response": result})
    31                except Exception as e:
    32                    print(f"Error processing query '{query}': {e}")
    33                    results.append({"query": query, "user_profile": user_profile, "response": {"error": str(e)}})
    34
    35            with open(output_file, "w", encoding="utf-8") as f:
    36                # Use ensure_ascii=False for proper display of non-ASCII characters
    37                json.dump(results, f, indent=2, ensure_ascii=False)
    38            print(f"Finished processing queries. Results saved to {output_file}")
    39
    40        if __name__ == "__main__":
    41            process_queries()

```

rica999513 (3 weeks ago) Ln 14, Col 17 Spaces:4 UTF-8 CRLF () Python base (3.12.7) ⓘ Go Live

Query_results.json

```

Richa_Mishra > complete_project > query_results.json > response > retrieved > meta
    [
        {
            "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, future steps: [], raw: (\n \\"eligibility_status\\": \"not eligible\", \n \\"reason\\\": \"The provided policy extracts detail visa requirements for Ireland,\n\n\", final_confidence: 0.8242857950925826, retrieval_score_mean: 0.4142859836419423, retrieved": [
                        {
                            "uid": "117",
                            "score": 0.43422904809316,
                            "vector_score": 0.50149285791380444,
                            "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. Financial Requirements / Proof of Means\n* No single EU-wide fixed bank balance* is mandated; consulates evaluate individual circumstances"
                        },
                        {
                            "uid": "151",
                            "score": 0.4187694748242696,
                            "vector_score": 0.4757269024848938,
                            "keyword_score": 0.3333333333333333,
                            "meta": [
                                {
                                    "source": "schengen_101.txt",
                                    "chunk_id": 151
                                }
                            ],
                            "text": "...# 3. Financial Requirements / Proof of Means\n* No single EU-wide fixed bank balance* is mandated; consulates evaluate individual circumstances"
                        }
                    ]
                }
            }
        }
    ]

```

rica999513 (3 weeks ago) Ln 33, Col 22 Spaces:2 UTF-8 CRLF () JSON ⓘ Go Live

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. Core Retrieval Components (rag/)

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.

👉 Pseudocode/Code Snippet:

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

```

13  class Retriever:
14      def retrieve(self, query: str, top_k: int = 5, query_embedding=None) -> List[Dict[str, Any]]:
15          results = []
16
17          # build keyword list from query
18          ql = query.lower()
19          keywords = [kw for kw in (self.financial_keywords + self.sponsorship_keywords + self.study_keywords) if kw.lower() in ql]
20
21          if query_embedding is not None:
22              qv = np.array(query_embedding, dtype="float32")
23              if qv.ndim == 1:
24                  qv = qv.reshape(1, -1)
25
26              # Normalize embedding vector
27              faiss.normalize_L2(qv)
28
29              # Search FAISS index
30              scores, ids = self.index.search(qv, top_k * 3)
31
32              # Combine vector search with keyword scoring
33              for vector_score, rid in zip(scores[0], ids[0]):
34                  if int(rid) < 0:
35                      continue
36
37                  sid = str(int(rid))
38
39                  meta = self.metadata.get(sid, {}) if isinstance(self.metadata, dict) else {}
40                  text = self.chunks.get(sid, "") if isinstance(self.chunks, dict) else ""
41
42                  kw_score = self._keyword_match_score(text, keywords) if keywords else 0.0
43
44                  # Combine scores: 60% vector similarity, 40% keyword match
45                  combined_score = float(vector_score) * 0.6 + kw_score * 0.4
46
47                  results.append({
48                      "uid": sid,
49                      "score": combined_score,
50                      "vector score": float(vector_score),
51                      "keyword score": kw_score,
52                      "meta": meta,
53                  })
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92

```

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

```

The screenshot shows a Jupyter Notebook interface with a single code cell containing Python code. The code defines a function `run_rag` that takes a query and top_k (5 by default) and returns a dictionary of results. It uses a stateless RAG pipeline with a retriever and a LLM (Gemini). The code handles errors, extracts information from the response, and maps 'yes/no' decisions. The notebook is titled "pipeline.py 1" and is part of a project named "complete_project". The right side of the interface shows a sidebar with various icons and a preview of the code.

```
File Edit Selection View Go Run Terminal Help ← → Q Swift_visa
```

```
Richa_Mishra > complete_project > rag > pipeline.py > run_rag
```

```
128
129 def run_rag(query: str, top_k: int = 5) -> Dict[str, Any]:
130     """
131     Fully stateless RAG pipeline for swiftvisa.
132     """
133     query_embedding = get_embedding(query)
134     retrieved = retriever.retrieve(query, top_k=top_k, query_embedding=query_embedding)
135     scores = [r.get("score", 0.0) for r in retrieved] if retrieved else []
136
137     # Check if retrieval confidence is too low to bother LLM
138     retrieval_conf = compute_confidence_from_scores(scores)
139     # The current threshold for model output filter is 0.514, keep confidence high
140
141     prompt = build_prompt(query, retrieved, max_chars=3000, mode="decision")
142
143     try:
144         raw_resp = call_gemini(prompt)
145     except Exception as e:
146         raw_resp = f"ERROR: LLM call failed outside of client wrapper: {e}"
147
148     parsed = extract_info(raw_resp)
149
150     # Compute blended confidence
151     declared_conf = parsed.get("confidence") or 0.0
152     try:
153         final_conf = 0.6 * retrieval_conf + 0.4 * float(declared_conf)
154     except Exception:
155         final_conf = retrieval_conf
156
157     # Map yes/no
158     dec = (parsed.get("decision") or "").lower()
159     yes_no = None
160     if "not eligible" in dec:
161         yes_no = 0
162     elif "eligible" in dec:
163         yes_no = 1
164     elif "partially" in dec:
```

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

The screenshot shows a code editor with two tabs open. The left tab contains the `logger.py` script, which handles logging visa decisions. The right tab contains a partially visible script, likely `visa_log.py`, which handles logging conversations.

```
logger.py
# File: logger.py
# Description: A script for logging visa decisions and conversations.

# Import required modules
import json
from datetime import datetime

# Define log_decision function
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 1,
            "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 if raw_prompt is not None else ""
        }
        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}")

# Define log_conversation function
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:
        print(f"[logger] Failed to write conversation log: {e}")

# Define constants
DECISION_LOG_FILE = "decisions.log"
CONVERSATION_LOG_FILE = "conversations.log"
```

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*

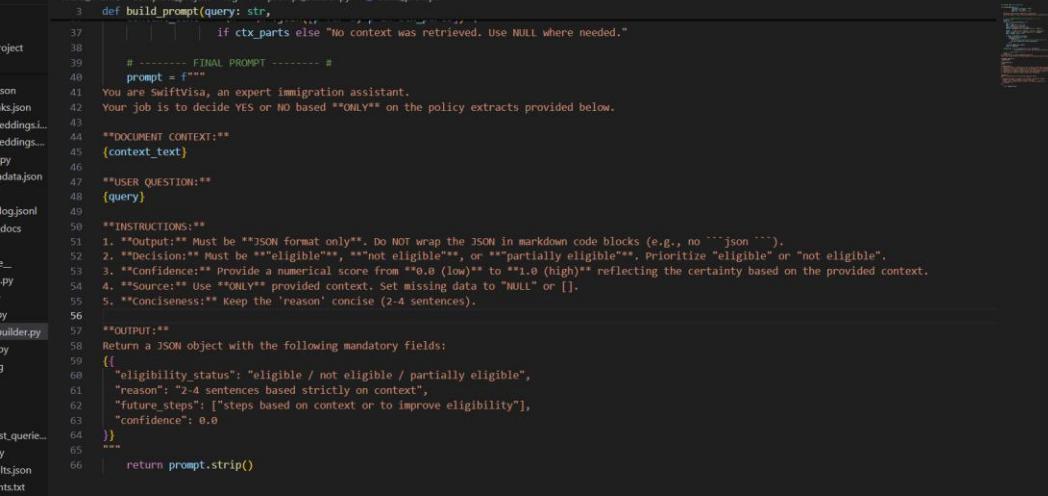
- **Role:** Builds structured prompts by combining the user query with retrieved chunks.

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(q, r, max, mode): RETURN final_prompt
```



```
richa999513 (3 weeks ago) Ln 56 Col 1 Spaces: 4 UTF-8 CRLF () Python base (3.12.7) ⌂ Go Live
```

The screenshot shows a terminal window with the following output:

```
Richa_Mishra@richa-Mishra:~/Documents/complete_project> python prompt_builder.py
```

b. *ilm_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

The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Search Bar:** Q Swift_visa
- Toolbar:** Includes icons for file operations, search, and help.
- Code Cell:** The cell contains Python code for calling the Gemini API. It includes imports from `llm_client.py` and defines functions for generating text and extracting responses.
- Output Cell:** Shows the result of running the code, indicating that the Gemini API call failed due to an error in the configuration.
- Bottom Status Bar:** Shows the user's name (Richa Mishra), the date (4 weeks ago), and the file path (llm_client.py).

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:
 - o Final answer.
 - o Top 5 related chunk IDs.
 - o Eligibility reasoning.
 - o Suggested future steps.
3. The response is shown either in:
 - o **Streamlit Dashboard** (streamlit_app.py) for interactive UI.
 - o **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:

The screenshot shows a VS Code interface with the following details:

- File Explorer:** Shows a project structure with files like `sample_queries.txt`, `pipeline.py`, `Richa_Mishra`, and `SWIFT_VISA`.
- Code Editor:** Displays `pipeline.py` which interacts with `nun_rag` and `run_rag` functions.
- Terminal:** Shows the command `python query_cli.py` being run in a terminal window titled "python - complete_project". The output indicates the user is partially eligible for an F-1 visa.
- Output:** Shows the result of running `query_cli.py`, detailing the user's eligibility status as "partially eligible" with a confidence score of 0.9.
- Problems:** Shows no errors or warnings.

Streamlit Dashboard Screen:

SwiftVisa AI - Your AI Eligibility Assistant

localhost:8501

Quick Guide

How to Use:

- Enter your query about visa eligibility
- Click analyze to get AI-powered results
- Review your eligibility status and recommendations

Supported Visa Types:

- Work Visas
- Family/Spouse Visas
- Student Visas
- Tourist Visas
- Investment Visas

Countries Covered:

United Kingdom, United States, Canada, Australia, European Union, and many more...

Tips for Best Results:

- Be specific about your situation
- Include relevant financial details
- Mention your relationship status

USA UK CANADA IRELAND SCHENGEN

SwiftVisa AI

Your Intelligent Visa Eligibility Assistant - Powered by AI

Instant Analysis

Get visa eligibility results in seconds with our advanced AI

Accurate Results

AI-powered analysis based on official immigration guidelines

Multiple Countries

Support for visa applications across various countries

Try Your Eligibility Check

How it works: Enter your visa query below and our AI will analyze your eligibility based on official immigration requirements and provide personalized guidance.

The screenshot shows the homepage of the SwiftVisa AI platform. At the top, there's a header bar with a back button, a search icon, a star icon, a user profile icon, and a 'Chat' button. On the right side of the header, there are 'Deploy' and three-dot menu icons. Below the header, the main content area has a title 'Try Your Eligibility Check' with a green leaf icon. A sub-section titled 'How it works:' explains that users can enter their visa query and receive personalized guidance. A text input field contains the query: 'I have a job offer but my CoS is delayed can I still apply for L1 visa?'. Below the input field is a blue button labeled 'Analyze My Eligibility'. A green success message at the bottom states 'Analysis completed successfully!'. At the bottom of the main content area, there's a section titled 'AI Assistant - Your Eligibility Analysis' with a microphone icon. It displays the following results:

- Eligibility Status: eligible
- Confidence Score: 0.0
- Reason for Decision: The provided policy extracts do not indicate that a Certificate of Sponsorship (CoS) is a requirement for an L1 visa.
- Actions to Improve: None

This screenshot shows the detailed analysis results. The top part is identical to the previous screenshot, displaying the AI Assistant analysis results. Below this, there's a section titled 'Analysis Metrics' with two large blue boxes. The left box is titled 'CONFIDENCE SCORE' and shows '47.9%' in large white text, with the note 'Based on 5 official sources' below it. The right box is titled 'DOCUMENTS ANALYZED' and shows the number '5' in large white text, with the note 'Official immigration guidelines' below it. At the bottom of the page, there's a section titled 'Referenced Documents' with a document icon. It lists several documents used for the analysis, each preceded by a small circular icon: Doc #226, Doc #310, Doc #350, Doc #348, Doc #347, and Doc #346. The entire page is framed by a dark border.

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