Project Report

On

Swift Visa AI- Based
Visa Eligibility Screening Agent

Submitted by

Dhanalaxmi Utkam

Project Statement:

Visa eligibility screening is a complex and policy-driven process that requires applicants to interpret detailed immigration rules, documentation requirements, and financial criteria that vary across countries and visa categories. Existing eligibility checking methods primarily rely on static rule-based systems, manual consultations, or unverified online sources, which often lead to inaccurate interpretations and inconsistent outcomes. These approaches lack contextual understanding and fail to provide transparent reasoning for eligibility decisions.

This project presents **SwiftVisa**, an intelligent visa eligibility screening system powered by Large Language Models (LLMs) and Retrieval-Augmented Generation (RAG). The proposed system evaluates user eligibility by analysing structured applicant information and grounding the decision in authoritative immigration policy documents. User inputs, including personal details, destination country, visa type, financial proof, and supporting documents, are semantically encoded and used to retrieve relevant policy information from a FAISS-based vector database.

The retrieved policy context is integrated with the user profile through a RAG framework to guide the reasoning process of the LLM, ensuring that eligibility decisions are policy-aligned and reducing the risk of hallucinated responses. The system generates a clear eligibility outcome, a confidence score representing decision certainty, and a concise explanation grounded in the retrieved policy content.

The solution is implemented using a Streamlit-based web interface, enabling interactive and real-time eligibility screening. The system avoids dependency on rule-based engines, optical character recognition, translation services, or live web scraping, thereby ensuring transparency, modularity, and controlled knowledge sources. SwiftVisa serves as a decision-support tool that assists users in understanding visa requirements and making informed application decisions, while not replacing official embassy or immigration authority decisions.

System Pipeline:

The SwiftVisa system follows a structured Retrieval-Augmented Generation (RAG) pipeline to ensure accurate, explainable, and policy-grounded visa eligibility decisions. The complete workflow is described as follows:

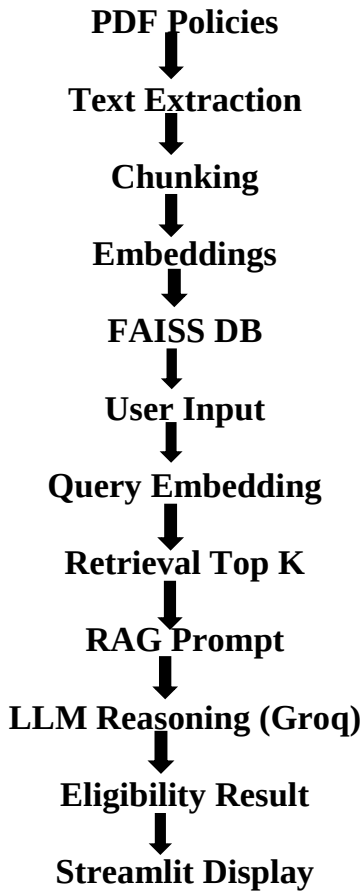


Fig: Swift Visa Screening Pipeline

1. Visa Policy Data Collection

Official visa policy documents are collected in **PDF format** from embassy or government websites. These documents contain eligibility rules, financial requirements, and document checklists.

2. PDF Text Extraction

The visa policy PDFs are processed to extract plain text.
This removes unnecessary formatting, images, and layout information while preserving policy content.

3. Text Cleaning and Chunking

The extracted text is:

- Cleaned to remove noise
- Split into small, meaningful chunks

Each chunk represents a specific visa rule or requirement.
This improves retrieval accuracy and model efficiency.

4. Embedding Generation

Each text chunk is converted into a vector embedding using a Sentence Transformer model.
These embeddings capture the semantic meaning of visa rules.

5. Vector Database Storage (FAISS)

All policy embeddings are stored in a FAISS vector database.
FAISS enables fast similarity search even for large datasets.

6. User Input Collection (Streamlit UI)

The user enters structured details through the Streamlit application, including:

- Personal details
- Destination country
- Visa type
- Financial proof
- Documents
- Case description

7. Query Embedding

The user input is combined into a single query and converted into an embedding vector using the same model used for policy data.

8. Semantic Retrieval

FAISS compares the user query embedding with stored policy embeddings and retrieves the top-K most relevant policy chunks.

9. RAG Prompt Construction

The retrieved policy chunks and user profile are combined to form a Retrieval-Augmented Generation (RAG) prompt.

This ensures the AI model reasons only from official policy information**.**

10. Eligibility Reasoning (LLM)

A Large Language Model processes the RAG prompt and performs logical reasoning to generate:

- Eligibility decision (YES / NO)
- Confidence score (0–1)
- Clear explanation

11. Result Presentation

The results are displayed in the Streamlit interface with:

- Eligibility status
- Confidence bar
- Reasoning explanation
- History of previous checks

12. Session History Storage

All results are temporarily stored in session memory, allowing users to review past eligibility checks.

METHODOLOGY:

1. Dataset Preparation

Dataset preparation is an essential phase in the SwiftVisa – Visa Eligibility Screening System. This phase focuses on collecting, cleaning, and structuring visa-related policy data so that it can be efficiently processed by the AI system in later stages.

1.1. Data Collection

Visa-related policy documents were collected from official and trustworthy sources, including government immigration portals, embassy websites, and consulate publications. These documents contain essential information such as eligibility conditions, required documents, financial requirements, and special visa rules.

All collected documents were stored in a centralized directory named Visa_Eligibility**.** This folder contains visa-related PDF documents for five countries: United States (US), United Kingdom (UK), Canada, Ireland, and the Schengen**.**

1.2. Data Format Identification

The majority of visa policy information is available in Portable Document Format (PDF). While PDFs are suitable for human reading, they are not directly usable for automated

processing. Therefore, an initial assessment of the document formats was conducted to determine suitable text extraction methods.

1.3. Text Extraction

Text content was extracted from the collected PDF documents using document processing techniques. During this process, all meaningful textual information such as headings, paragraphs, and bullet points was retained. Non-relevant elements including images, logos, page numbers, and decorative formatting were excluded to obtain clean textual data.

1.4. Data Cleaning and Normalization

The extracted text often contained inconsistencies such as extra whitespace, broken sentences, repeated headers, and formatting artifacts. To address these issues, a data cleaning process was applied. This process involved removing unnecessary symbols, normalizing punctuation, correcting spacing, and ensuring uniform text formatting across all documents.

1.5. Document Segmentation

Due to the large size of visa policy documents, the cleaned text was divided into smaller and meaningful segments known as text chunks. Each chunk represents a specific visa rule or policy condition and is structured to preserve contextual meaning. This segmentation improves information accessibility and supports efficient retrieval in later stages.

1.6. Chunk Organization and Storage

Each segmented text chunk was saved as an individual text file with a unique identifier. All chunk files were organized within a dedicated directory structure. This approach enables systematic data management, easy referencing, and traceability during system processing.

```
import os
import re
import pdfplumber
import numpy as np


def preprocess_pdf(pdf_path, output_dir="cleaned_texts"):

    os.makedirs(output_dir, exist_ok=True)

    print(f"[INFO] Extracting from: {pdf_path}")

    with pdfplumber.open(pdf_path) as pdf:
        text = "\n".join(page.extract_text() or "" for page in pdf.pages)
```

```
 # Cleaning text
    text = re.sub(r'\s+', ' ', text)           # remove extra spaces
    text = re.sub(r'[^a-zA-Z0-9.,;:?!()/%\-\n ]', '', text)  # remove junk
chars
    text = text.lower().strip()

    file_name = os.path.splitext(os.path.basename(pdf_path))[0] +
"_cleaned.txt"
    cleaned_path = os.path.join(output_dir, file_name)

    with open(cleaned_path, "w", encoding="utf-8") as f:
        f.write(text)

    print(f"[INFO] Cleaned saved: {cleaned_path}")
    return cleaned_path
# CHUNKER (Split cleaned text into small chunks)

def chunk_text(cleaned_file, max_length=300, overlap=100,
output_dir="chunks"):
    """
    Creates multiple small chunks from a cleaned text file.
    """
    os.makedirs(output_dir, exist_ok=True)

    with open(cleaned_file, "r", encoding="utf-8") as f:
        text = f.read()

    words = text.split()
    chunks = []
    chunk = []
    for word in words:
        chunk.append(word)
        if len(chunk) >= max_length:   # 300 words per chunk
            chunks.append(" ".join(chunk))
            chunk = []
    if chunk:
        chunks.append(" ".join(chunk))

    base = os.path.splitext(os.path.basename(cleaned_file))[0]
    chunk_files = []
    for i, c in enumerate(chunks):
        path = os.path.join(output_dir, f"{base}_chunk{i}.txt")
        with open(path, "w", encoding="utf-8") as f:
            f.write(c)
        chunk_files.append(path)

    print(f"[INFO] {len(chunk_files)} chunks created for {cleaned_file}")
    return chunk_files
```

The dataset preparation phase transforms raw visa policy documents into a structured and high-quality textual dataset. This process plays a vital role in enabling accurate policy-based analysis and decision-making in the proposed system.

2. Embeddings Generation

2.1. Introduction

Embedding generation is a crucial step in the SwiftVisa system. It converts visa policy text into numerical form so that the system can understand, compare, and retrieve relevant information efficiently. Without embeddings, the system would rely only on keyword matching, which is slow and inaccurate for complex visa queries.

What Are Embeddings?

Embeddings are **numerical vector representations** of text that capture its **meaning**, not just the words used.

- Each sentence or paragraph is converted into a fixed-length vector.
- Texts with similar meaning produce vectors that are close to each other in vector space.
- This allows semantic search instead of exact keyword search.

For example:

- "Proof of funds is required"
- "Bank balance must be shown"

Both sentences have different words but similar meaning, and their embeddings will be close.

Why Embeddings Are Used in SwiftVisa

Visa policies are written in formal and complex language. Users often ask questions in simple or informal language. Embeddings bridge this gap by understanding **semantic similarity** between user queries and policy documents.

Key benefits:

- Improves accuracy of document retrieval
- Handles synonyms and paraphrased queries
- Enables faster search using vector databases
- Works well with AI models in RAG systems

2.2. Embedding Model Used

SwiftVisa uses a Sentence Transformer model**.**

SwiftVisa employs a pre-trained sentence-level embedding model based on transformer architecture. The model is trained on large-scale textual data and is capable of understanding contextual relationships between words and sentences. It produces dense vector representations of fixed dimensionality for any given input text.

The selected embedding model is optimized for:

- Sentence similarity tasks
- Question–answer matching
- Information retrieval applications

2.3. Embedding Generation Process

The embedding generation process consists of the following stages:

1. **Input Preparation**
   Each text chunk obtained during the document chunking phase is treated as an independent input unit. These chunks typically contain eligibility criteria, financial requirements, document lists, and visa-specific conditions.
2. **Text Encoding**
   The embedding model processes each input chunk by analysing lexical patterns, syntactic structure, and contextual dependencies. Through multiple transformer layers, the model encodes semantic information into numerical form.
3. **Vector Representation**
   The output of the embedding model is a dense numerical vector that represents the semantic meaning of the input text. Text segments conveying similar concepts result in vectors that are closer in the embedding space.

```
#  EMBEDDING CREATOR
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("all-MiniLM-L6-v2")

def embed_chunks(chunk_files):

    embeddings = []
    texts = []

    print("\n===== GENERATING EMBEDDINGS =====")

    for file in chunk_files:
        with open(file, "r", encoding="utf-8") as f:
            text = f.read()

        texts.append(text)
        # Generate embedding for current chunk
        embedding = model.encode(text)
        embedding_np = np.array(embedding)
```

```
        print(f"\n[EMBEDDING GENERATED] File: {file}")
        #print(f"Vector (first 10 values): {embedding_np[:10]}")
        print(f"Shape: {embedding_np.shape}")
        # Append ONLY the current embedding
        embeddings.append(embedding_np)

    # Convert full list to array
    all_embeddings_np = np.array(embeddings)

    print("\n===== FINAL EMBEDDINGS SHAPE =====")
    print(all_embeddings_np.shape)
    # Save once
    np.save("visa_embeddings.npy", all_embeddings_np)
    print("[INFO] Saved all embeddings to visa_embeddings.npy")

    return all_embeddings_np, texts
```

2.4. Query Embedding

To ensure consistency, user queries undergo the same embedding generation process as policy text chunks. This allows direct comparison between query embeddings and document embeddings within the same semantic space. As a result, the system can identify policy sections that are conceptually aligned with the user's input, even when different terminology is used.

2.5. Role in Retrieval-Augmented Generation

Embedding generation plays a critical role in the Retrieval-Augmented Generation (RAG) architecture adopted by SwiftVisa. The generated embeddings facilitate the selection of relevant policy content, which is then provided as contextual input to a large language model. This ensures that eligibility decisions and explanations are grounded in official policy information rather than generic knowledge.

2.6. Advantages of Embedding-Based Representation

The use of embeddings offers several advantages over traditional keyword-based approaches:

- Improved semantic understanding of text
- Robust handling of synonyms and paraphrased queries
- Reduced dependency on exact word matches

Enhanced retrieval accuracy for complex policy documents

Embedding generation enables SwiftVisa to convert textual visa policy data into meaningful numerical representations that support semantic retrieval and intelligent decision-making. By aligning user queries and policy content within a shared embedding space, the system achieves higher accuracy, efficiency, and reliability in visa eligibility assessment.

3. FAISS Database Storage

FAISS (Facebook AI Similarity Search) is used in the SwiftVisa system to efficiently store and retrieve high-dimensional vector representations of visa policy documents. It enables fast similarity-based search, which is critical when working with large textual datasets.

3.1. Purpose of FAISS in SwiftVisa

The primary purpose of using FAISS is to support efficient semantic retrieval**.** Instead of comparing user queries with all policy documents directly, FAISS allows the system to quickly identify the most relevant policy chunks based on vector similarity.

FAISS is designed for:

- Fast nearest-neighbour search
- Handling large-scale vector data
- Low-latency retrieval during real-time inference

3.2. Data Stored in FAISS

FAISS stores numerical vector embeddings**,** not raw text. In SwiftVisa:

- Each visa policy document is first divided into smaller text chunks
- Each chunk is converted into a fixed-length vector using a sentence embedding model
- These vectors are stored inside the FAISS index

The original text chunks are stored separately and retrieved only after FAISS returns the most relevant vector indices.

3.3. Index Construction Process

The FAISS storage process follows these steps:

1. **Embedding Generation**
   Each policy text chunk is transformed into a dense numerical vector.
2. **Index Initialization**
   A FAISS index is created based on the vector dimensionality.
3. **Vector Insertion**
   All embedding vectors are added to the FAISS index.
4. **Persistent Storage**
   The index is saved to disk, allowing reuse without rebuilding during every system run.

This approach significantly reduces system startup time.

3.4. Similarity Search Mechanism

When a user submits a query:

1. The query text is converted into an embedding vector
2. FAISS compares this vector with stored policy vectors
3. Distance metrics (e.g., cosine similarity or L2 distance) are used
4. The top-K closest vectors are returned

These vectors correspond to the most relevant policy chunks.

```
#VECTOR DATABASE (FAISS)
def store_faiss(embeddings, index_path="visa_index.faiss"):
    """
    Saves embeddings to a FAISS vector index.
    """
    embeddings = embeddings.astype("float32")

    index = faiss.IndexFlatL2(embeddings.shape[1])
    index.add(embeddings)

    faiss.write_index(index, index_path)

    print(f"[INFO] FAISS index created at: {index_path}")
    return index_path
```

3.5. Advantages of FAISS Storage

Using FAISS provides several advantages:

- ✓ High-speed similarity search even with thousands of documents
- ✓ Scalable to large datasets
- ✓ Optimized memory usage
- ✓ Real-time retrieval performance
- ✓ Easy integration with embedding models

3.6. Role in the RAG Pipeline

FAISS acts as the **retrieval backbone** of the Retrieval-Augmented Generation pipeline:

- It ensures that only relevant policy information is passed to the language model
- Reduces hallucination by grounding responses in actual policy text
- Improves decision accuracy and confidence scoring

FAISS enables efficient storage and retrieval of semantic embeddings in SwiftVisa. By separating vector search from language generation, the system achieves fast, scalable, and accurate visa eligibility screening.

4. Pipeline Execution for Multiple Visa Policy PDFs

4.1. Purpose of the Pipeline Runner

The pipeline runner is designed to process multiple visa policy PDF documents in a single execution. This enables the system to handle visa rules from different countries efficiently and store them in a unified vector database while maintaining traceability between document chunks and their corresponding countries.

4.2. Function Overview

The function **run_pipeline_for_folder(pdf_folder)** automates the complete preprocessing and embedding workflow for all PDF files located in a specified directory.

Its responsibilities include:

- Reading multiple visa policy PDFs
- Preprocessing and cleaning text
- Chunking documents into smaller units
- Generating embeddings
- Storing embeddings in FAISS
- Maintaining a mapping between countries and chunk indices

```
#  PIPELINE RUNNER FOR MULTIPLE PDFs
def run_pipeline_for_folder(pdf_folder):
    pdf_files = [os.path.join(pdf_folder, f) for f in os.listdir(pdf_folder)
                 if f.lower().endswith(".pdf")]
    all_chunk_files = []
    index_map = {}  # NEW: country → (start_index, end_index)
    current_index = 0
    for pdf in pdf_files:
        print(f"\n--- Processing: {pdf} ---")
        cleaned = preprocess_pdf(pdf)
        chunks = chunk_text(cleaned)
        all_chunk_files.extend(chunks)
        # Save the range for this country
        country = os.path.splitext(os.path.basename(pdf))[0]  # file name
without extension
        index_map[country] = (current_index, current_index + len(chunks) - 1)
        current_index += len(chunks)
    # Embeddings
    print(f"\n[INFO] Total chunks from all PDFs: {len(all_chunk_files)}")
    embeddings, chunk_texts = embed_chunks(all_chunk_files)
    index_file = store_faiss(embeddings)
    print("\n===== PIPELINE COMPLETED FOR ALL PDFs =====")
    # Save index map for future querying
    np.save("country_index_map.npy", index_map)

    return index_file, chunk_texts, index_map
```

4.3. Input Handling

The function accepts a single input parameter:

- pdf_folder**:** A directory containing visa policy documents in PDF format. Each PDF represents the visa policies of a specific country.

The function scans this folder and filters all files ending with the .pdf extension.

4.4. Chunk Collection and Index Mapping

For each PDF document:

1. **Preprocessing**
   The PDF text is extracted and cleaned to remove noise such as headers, footers, and unnecessary symbols.
2. **Chunking**
   The cleaned text is divided into smaller chunks to ensure better semantic representation during embedding.
3. **Chunk Aggregation**
   All chunks from all PDFs are stored in a single list (all_chunk_files).
4. **Country Index Mapping**
   A dictionary (index_map) is maintained to track the range of chunk indices belonging to each country.

   This mapping records:

   - Starting index of the country's chunks
   - Ending index of the country's chunks

This mapping allows the system to later identify which country a retrieved chunk belongs to.

4.5. Embedding Generation

After all PDFs are processed:

- The complete list of chunks is passed to the embedding function.
- Each chunk is converted into a dense numerical vector using a sentence embedding model.
- The resulting embeddings preserve the semantic meaning of the visa policy text.

4.6. FAISS Index Storage

The generated embeddings are stored in a FAISS index:

- The FAISS index enables fast similarity search
- The index is saved to disk for reuse
- This avoids recomputing embeddings during every query execution

4.7. Metadata Persistence

To maintain country-level traceability:

- The country-to-index mapping is saved as a NumPy file (country_index_map.npy)
- This file is later used during retrieval to filter or interpret search results by country

4.8. Main Execution Logic

The __main__ block specifies the folder containing visa policy PDFs and invokes the pipeline function.

```
if __name__ == "__main__":
    folder = "Visa_Eligibility"   # folder containing  visa PDFs

    run_pipeline_for_folder(folder)
```

This ensures:

- The entire pipeline runs automatically for all PDFs
- The system is scalable to new countries by simply adding new PDF files

4.9. Advantages of the Multi-PDF Pipeline

- Supports multiple countries in a single vector database
- Maintains clear mapping between chunks and countries
- Improves scalability and maintainability
- Enables accurate country-specific retrieval during inference

The pipeline runner provides a structured and automated approach for processing multiple visa policy documents. By integrating preprocessing, chunking, embedding generation, FAISS storage, and country-level indexing, the system ensures efficient and accurate retrieval for visa eligibility screening.

5. Top-K Documents Retrieval

The document retrieval module is responsible for retrieving the **most relevant visa policy documents** based on a user query. This is a crucial part of the SwiftVisa system, forming the first step in a **Retrieval-Augmented Generation (RAG) pipeline**, where retrieved documents are later analyzed by an AI model to provide an eligibility decision

5.1. Configuration and Data

- **Chunks Folder**: Directory containing preprocessed document chunks. Each chunk represents a small portion of a visa policy document.
- **FAISS Index File**: Stores the vector indices of all document embeddings for fast similarity search.
- **Embeddings File**: Stores precomputed embeddings for all document chunks.

- **Embedding Model**: The pre-trained SentenceTransformer model used to encode queries.
- **TOP_K**: The number of most relevant documents to retrieve per query (e.g., top 4).

```
CHUNKS_FOLDER = "chunks"
INDEX_FILE = "visa_index.faiss"
EMBEDDINGS_FILE = "visa_embeddings.npy"
EMBEDDING_MODEL = "all-MiniLM-L6-v2"
TOP_K = 4
```

5.2 Functionality of the Retrieval Process

1. **Loading the Embedding Model**:
    - The model is loaded into memory to encode the user query into a vector embedding in the same vector space as the documents.
2. **Loading FAISS Index and Embeddings**:
    - The FAISS index provides an efficient way to find the nearest document embeddings to the query vector.
    - Precomputed embeddings are loaded to ensure consistency with the index.
3. **Listing Document Chunks**:
    - Document filenames are listed in a sorted order to **map indices returned by FAISS to actual files** reliably.
4. **Query Encoding**:
    - The user's text query is converted into a **dense vector embedding** representing its semantic meaning.
5. **FAISS Similarity Search**:
    - The query embedding is compared with all document embeddings.
    - The FAISS index returns the **indices of the top-K most similar document chunks**.
6. **Mapping to Filenames**:
    - FAISS indices are mapped back to document filenames to provide the **actual retrieved documents** that will be used in the next step of the RAG pipeline.

5.3 Purpose and Importance

- **Semantic Retrieval**: Unlike keyword matching, this module retrieves documents based on **semantic similarity**, capturing the meaning of the query.

- **Efficiency**: FAISS allows **fast searches**, even with large numbers of documents.
- **Chunk-Based Approach**: Documents are divided into smaller chunks to improve **granularity and precision** of retrieval.
- **Integration with RAG**: The retrieved chunks serve as **context for the AI model**, ensuring accurate and informed eligibility decisions.

5.4. Example Workflow

- **User Query**: `"Can a student from India apply for a tourist visa to Canada?"`
- **Retrieved Chunks**:
  - Policy on Canadian tourist visa requirements
  - Financial proof rules
  - Age restrictions
  - Student visa eligibility criteria

These chunks are then passed to the AI decision engine, which evaluates the information and provides a structured eligibility decision.

```
def retrieve_top_k_documents(query):
    # Load embedding model
    model = SentenceTransformer(EMBEDDING_MODEL)

    # Load FAISS index + embeddings
    index = faiss.read_index(INDEX_FILE)
    embeddings = np.load(EMBEDDINGS_FILE)

    # List chunk text files in predefined order
    chunk_files = sorted([f for f in os.listdir(CHUNKS_FOLDER) if
f.endswith(".txt")])

    # Encode query
    q_emb = model.encode([query], convert_to_numpy=True).astype("float32")

    # FAISS similarity search
    _, indices = index.search(q_emb, TOP_K)

    # Return the retrieved chunk filenames
    return [chunk_files[i] for i in indices[0]]


if __name__ == "__main__":
    query = input("Enter visa question:\n> ")

    # Task 1 → retrieve documents
    retrieved = retrieve_top_k_documents(query)
    print("\nRetrieved document chunks:", retrieved)
```

The document retrieval module ensures that the system provides **relevant, accurate, and up-to-date policy information**, forming the foundation for reliable AI-based visa screening. Its combination of embedding-based semantic search and FAISS indexing enables both **speed and precision**, essential for a real-time application like SwiftVisa.

6. AI-Powered Visa Eligibility Checking using LLM and RAG

The AI-powered visa eligibility checking module constitutes the core intelligence of the SwiftVisa system. It integrates a **large language model (LLM)** with **Retrieval-Augmented Generation (RAG)** to provide precise, evidence-based visa eligibility decisions. The approach ensures that user queries are evaluated in the context of **official visa policy documents**, improving both accuracy and transparency.

6.1. Libraries and Tools

The module relies on the following components:

1. **Groq SDK** – Provides an interface to the LLM API for prompt-based reasoning and structured response generation.
2. **Custom Storage Module** – Saves eligibility results in JSON format for maintaining a query history and supporting auditability.
3. **Retrieval Module** – Supplies the top-K relevant policy document chunks to the LLM for contextual analysis.

6.2. Input and Context Preparation

User input includes personal information, visa type, submitted documents, financial proof, and other relevant details. The retrieval module identifies the most relevant document chunks from a preprocessed corpus. These chunks are concatenated to form a **policy context**, which is provided as input to the LLM. By restricting the LLM to this context, decisions are based exclusively on validated policy information, mitigating hallucinations.

6.3. Prompt Design for LLM

- A carefully engineered prompt instructs the LLM to:
  1. Assess visa eligibility strictly using the provided policy extracts.
  2. Return a **YES/NO decision**.
  3. Assign a **confidence score** between 0.0 and 1.0, reflecting the strength of supporting evidence.
  4. Provide a concise **reasoning paragraph**.
  5. List **missing documents**, if any.
  6. Include **citations** of the document chunks used.
- This structured prompt ensures **consistency, transparency, and interpretability** of the AI's outputs.

6.4. LLM Evaluation and Structured Response

The LLM processes the query in the context of retrieved policy documents and generates a structured response containing:

- **Eligibility:** YES or NO
- **Confidence Score:** 0.0 to 1.0
- **Reason:** Short explanatory paragraph
- **Missing Documents:** List or NONE
- **Citations:** Chunk filenames used

The structured format allows for **easy parsing, display, and storage** of results in downstream systems.

```
from groq import Groq
client = Groq(api_key="Your_API_Key")
MODEL = "llama-3.1-8b-instant"

def ask_eligibility_decision(query, retrieved_chunks):
    # Build context from retrieved visa policy chunks
    context = ""
    for chunk_file in retrieved_chunks:
        with open(f"chunks/{chunk_file}", "r", encoding="utf-8") as f:
            context += f.read() + "\n\n"

    prompt = f"""
You are a visa eligibility decision engine.
Your job is to decide YES or NO based ONLY on the policy extracts provided
below.

USER INPUT:
{query}

POLICY INFORMATION:
{context}

You must evaluate confidence strictly by the amount of supporting evidence in
the policy documents.
Return the result in this EXACT format:

Eligibility: YES or NO
Confidence Score (0.0 to 1.0): <decimal>
Reason: short paragraph
Missing Documents (if any): list or say NONE
Citations: chunk filenames used
"""
    response = client.chat.completions.create(
        model=MODEL,
        messages=[{"role": "user", "content": prompt}],
        temperature=0.0
    )
    return response.choices[0].message.content
```

```
from retrieval import retrieve_top_k_documents

print("\n✨ SwiftVisa – Visa Eligibility Checker\n")

while True:
    query = input("Enter your profile and visa question:\n> ")

    retrieved = retrieve_top_k_documents(query)
    print("\nRetrieved document chunks:", retrieved)

    result = ask_eligibility_decision(query, retrieved)

    print("\n========== ELIGIBILITY RESULT ==========")
    print(result)
    print("========================================\n")

    # Save result
    save_to_json(query, result)

    again = input("Do you want to check another eligibility? (yes/no):
").strip().lower()
    if again not in ["yes","y"]:
        print("\n👋 Exiting SwiftVisa. Have a great day!")
        break
```

6.5. Workflow Summary

1. User submits a query describing personal profile and visa requirements.
2. Retrieval module selects the top-K relevant policy chunks.
3. LLM evaluates the query within the context of these chunks.
4. A structured eligibility result is generated and displayed.
5. The result is stored for history and auditing.

6.6. Advantages and Significance

- ✓ **Evidence-Based Decisions:** Evaluates queries strictly using retrieved policy documents.
- ✓ **Structured Output:** Provides eligibility, confidence, reason, missing documents, and citations for transparency.
- ✓ **Scalability and Interactivity:** Supports multiple queries in real-time.
- ✓ **Integration with RAG:** Combines document retrieval and AI reasoning to enable **context-aware decisions**.
- ✓ **Traceability:** Cited document chunks enhance accountability and trust in the AI system.

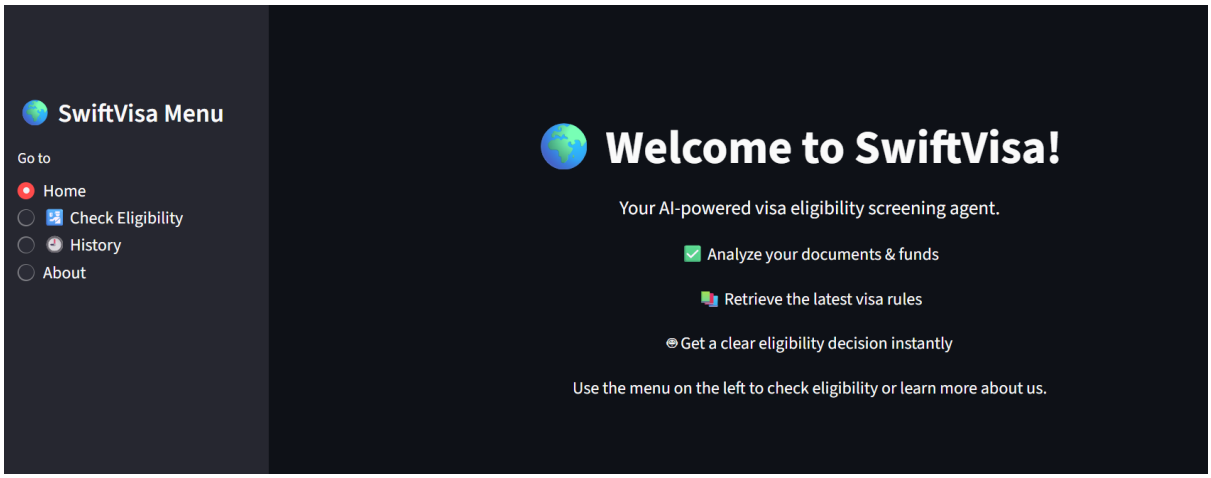
7. Streamlit Application

The Streamlit application serves as the **front-end interface** for the SwiftVisa system, integrating the **document retrieval** and **AI-based eligibility decision** modules into an interactive platform. It provides users with a simple, intuitive, and real-time experience for checking visa eligibility.

7.1. Page Layout and Navigation

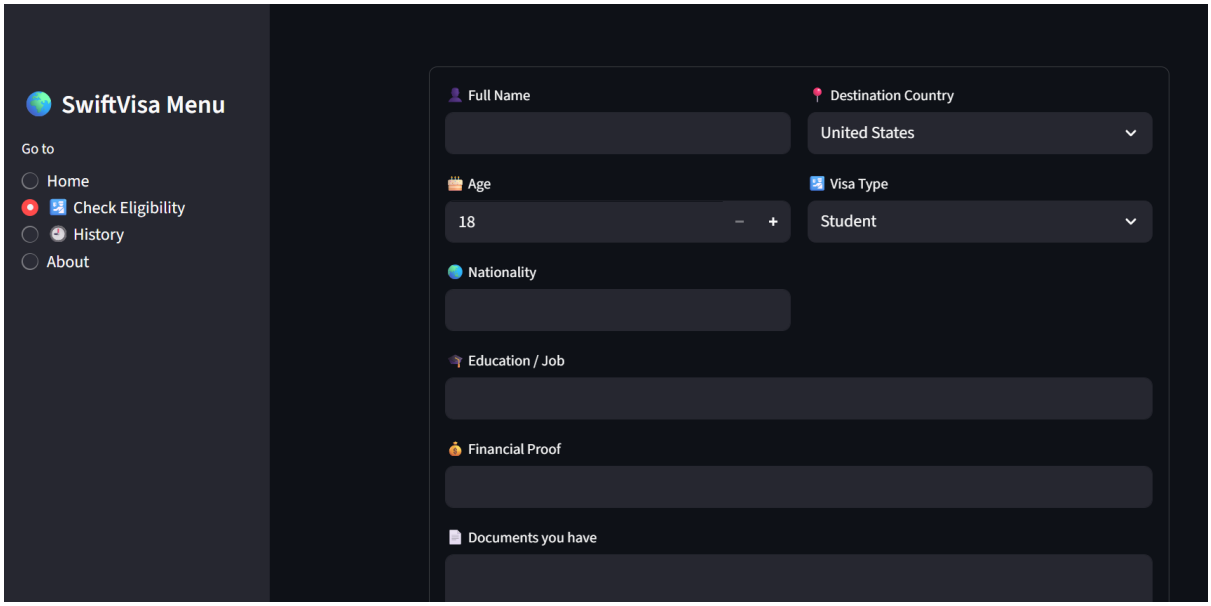
The application adopts a **multi-page structure** with a sidebar menu, allowing seamless navigation between different functionalities:

1. **Home Page:**
    - Welcomes users and provides an overview of SwiftVisa's capabilities.
    - Highlights key features: document analysis, policy retrieval, and instant eligibility decisions.
    - Guides users to the eligibility check form.



2. **Eligibility Check Page:**

    - Provides a structured form to collect user information, including personal details, destination country, visa type, educational background, financial proof, documents, and case description.
    - On form submission, the user query is processed by the **RAG + LLM pipeline**, producing a structured visa eligibility decision.

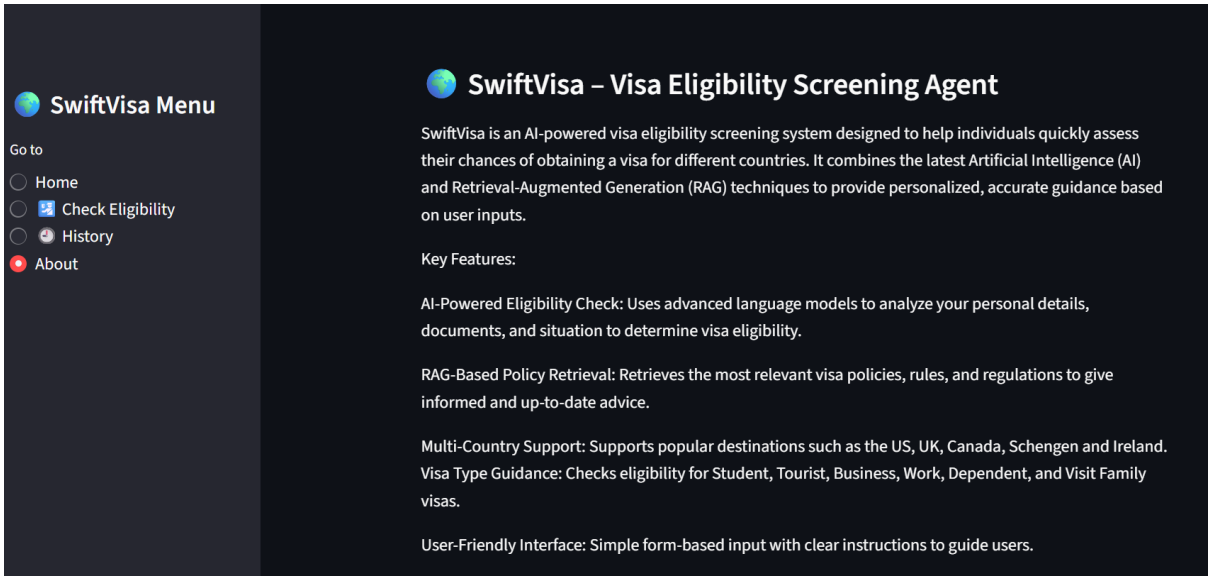


3. **History Page:**

   - Displays all previously submitted eligibility checks along with results, including eligibility status, confidence scores, reasons, and cited documents.
   - Enables users to review prior checks and maintain an audit trail.

4. **About Page:**

   - Explains the system's purpose, workflow, and AI-powered features.
   - Describes how RAG-based retrieval and LLM reasoning are combined to provide evidence-based visa decisions.



7.2. Integration with AI Modules

1. **Query Handling:**
    - User input is converted into a structured textual query including all relevant personal and visa-related details.
2. **Document Retrieval:**
    - The query is passed to the retrieval module, which returns the top-K most relevant visa policy document chunks using embeddings and FAISS similarity search.
3. **Eligibility Decision:**
    - Retrieved chunks are passed to the LLM-based decision engine.
    - The engine evaluates eligibility, calculates a confidence score, identifies missing documents, provides reasoning, and cites relevant policy chunks.
4. **Result Display:**

- The eligibility decision is displayed in an intuitive format, including:
    - Approval/rejection status with visual indicators.
    - Confidence score represented as both a numeric metric and a progress bar.
    - Detailed explanation of reasoning.

5. **Session Management:**
    - User queries and results are stored in the session state, allowing real-time tracking of multiple eligibility checks in a single session.

7.3. Features and Functionalities

- **Interactive Form-Based Input:** Collects user information efficiently.
- **Real-Time AI Decision:** Provides eligibility results immediately.
- **Structured and Transparent Output:** Includes eligibility, confidence, reasoning, missing documents, and citations.
- **History Tracking:** Allows users to view previous queries and results.
- **Multi-Country and Multi-Visa Support:** Covers popular visa destinations and types.
- **Secure and Private:** No permanent storage of sensitive user data; session-based processing ensures privacy.

7.4. Workflow Summary

1. User selects the **Eligibility Check** page and completes the form.
2. Input query is processed by the **document retrieval module**, retrieving the most relevant policy chunks.
3. The **LLM-based decision engine** evaluates eligibility using the retrieved context.
4. Results are displayed on the interface with clear indicators and metrics.
5. All results are logged in session history for future reference.

7.5. Significance and Advantages

- **User-Friendly Interface:** Simplifies interaction with a complex AI system.
- **Evidence-Based Decision Making:** Decisions are based on retrieved policy documents, ensuring accuracy and reliability.
- **Transparency and Traceability:** Structured output and document citations build trust.
- **Interactive and Scalable:** Supports multiple queries per session and is easily extendable for new countries or visa types.
- **Integration of RAG + LLM Modules:** Ensures that AI decisions are context-aware, policy-compliant, and actionable**.**