INTERNSHIP REPORT

SwiftVisa

Visa Eligibility Screening Agent

Created By

Shreyash Kumar

1. Introduction

This internship project focuses on building **SwiftVisa**, an AI-powered Visa Eligibility Screening Agent using a Retrieval-Augmented Generation (RAG) pipeline. The system is designed to assist users in understanding visa eligibility, requirements, and documentation by grounding Large Language Model (LLM) responses according to official immigration policy documents.

The project demonstrates the practical application of Natural Language Processing (NLP), embeddings, vector databases, and Large Language Models in a real-world, compliance-driven domain.

2. Project Objectives

- Build an end-to-end RAG pipeline for visa screening
- Process and index official immigration PDF documents
- Retrieve relevant policy information using semantic search
- Generate grounded and explainable visa eligibility responses
- Provide confidence scores and source citations

3. System Architecture Overview

Pipeline Flow:

PDF Documents → Text Extraction → Cleaning → Chunking → Embeddings → Vector Database (FAISS/Chroma) → Query Retrieval → LLM Response Generation → Confidence Scoring

4. Dataset Preparation

4.1 Document Collection

Official immigration documents were manually collected for the following countries:

- Canada
- Germany

- USA
- United Kingdom
- Ireland
- Schengen Countries

The dataset included:

- Government-issued visa policy guides
- Eligibility criteria documents
- Application checklists
- Work permit instructions
- Country-specific PR guidelines

4.2 Text Extraction

PDFs were processed using PDF text extraction libraries such as `pdfplumber` and `PyPDF2` to extract raw textual content from the collected documents.

Pseudo Code:

```
import os
import re
import json
import argparse
from pathlib import Path
from collections import Counter
import pdfplumber
import PyPDF2
import numpy as np
from tqdm import tqdm

# ---- NLP / Embedding libs ----
import spacy
from sentence_transformers import SentenceTransformer

# ---- FAISS (safe import with fallback) ----
try:
    import faiss
    print("FAISS successfully imported.")
```

Figure 1: Pseudo Code Representation

5. Text Cleaning & Preprocessing

The extracted text was cleaned using the following steps:

- Removal of headers and footers
- Normalization of whitespace
- Elimination of empty lines

Pseudo Code:

```
import re
from pathlib import Path
import pdfplumber
import PyPDF2
from collections import Counter
import pandas as pd
import os


def extract_text_from_pdf(path: str | Path):
    page_texts = []
    try:
        with pdfplumber.open(str(path)) as pdf:
            for p in pdf.pages:
                page_texts.append(p.extract_text() or "")
    except Exception:
        # fallback
        reader = PyPDF2.PdfReader(str(path))
        for p in reader.pages:
            try:
                page_texts.append(p.extract_text() or "")
            except Exception:
                page_texts.append("")
    return page_texts


def dehyphenate(text: str):
    return re.sub(r'(\w+)-\n(\w+)', r'\1\2', text)


def normalize_text(text: str):
    text = text.replace('\r', '\n')
    text = re.sub(r'\n{2,}', '\n\n', text)
    text = text.strip()
    return text
```

Figure 2: Pseudo Code Representation

6. Text Chunking Strategy

To enable efficient semantic retrieval, documents were divided into overlapping chunks:

- Chunk size: 400–800 tokens
- Overlap: 100–150 tokens

Pseudo Code:

```
# ============================================================
# 2. DOCUMENT CHUNKING
# ============================================================

def chunk_text(text, chunk_size=600, overlap=100):
    """Split large documents into overlapping chunks."""
    words = text.split()
    chunks = []

    start = 0
    while start < len(words):
        end = start + chunk_size
        chunk_words = words[start:end]
        chunks.append(" ".join(chunk_words))
        start = end - overlap

    return chunks
```

Figure 3: Pseudo Code Representation

7. Embedding Generation

Each text chunk was transformed into a numerical vector using embedding models.

7.1 Embedding Options Used

- **Sentence Transformers (BERT-based):** BERT-based models fine-tuned for semantic similarity. Commonly used model: `all-MiniLM-L6-v2`.
- **OpenAI Embeddings:** Proprietary embeddings optimized for semantic retrieval, classification, and clustering. Models used include `text-embedding-3-small` and `text-embedding-3-large`.
- **Gemini Text Embeddings:** Google's embedding models optimized for policy and structured document understanding.

7.2 Comparison of Embedding Models

Feature	Sentence Transformers	OpenAI Embeddings	Gemini Embeddings
Model Type	BERT-based	Proprietary	Gemini Transformer
Retrieval Accuracy	Good	Excellent	Very Good
Speed	Fast	Medium	Fast
Integration	FAISS, Chroma	FAISS, Chroma	FAISS
Best For	Local RAG	Production RAG	Cloud-scale

Pseudo Code:

```
# ============================================================
# 3. GEMINI EMBEDDING FUNCTION
# ============================================================

def embed_text(text: str):
    """Generate embeddings using Gemini text-embedding-004."""
    result = genai.embed_content(
        model="models/text-embedding-004",
        content=text
    )
    return np.array(result["embedding"], dtype="float32")
```

Figure 4: Pseudo Code Representation

8. Vector Database Implementation

A vector database was implemented to support fast semantic similarity search.

8.1 Databases Explored

- FAISS (preferred for speed and accuracy)
- ChromaDB

Similarity Metric: Cosine Similarity

Pseudo Code:

```python
# ============================================================
# 5. LOAD OR BUILD FAISS INDEX
# ============================================================

def load_faiss_index(folder="faiss_index"):
    index_file = os.path.join(folder, "index.faiss")
    docs_file = os.path.join(folder, "docs.txt")


    if os.path.exists(index_file) and os.path.exists(docs_file):

        print("🔹 Loading existing FAISS index...")

        index = faiss.read_index(index_file)

        with open(docs_file, "r", encoding="utf-8") as f:
            docs_raw = f.read().split("\n-----\n")

        docstore = InMemoryDocstore()
        for i, text in enumerate(docs_raw):
            docstore.add({str(i): Document(page_content=text)})

        vector_store = FAISS(
            embedding_function=embed_text,
            index=index,
            docstore=docstore,
            index_to_docstore_id={i: str(i) for i in range(len(docs_raw))}
        )

        return vector_store
```

Figure 5: Pseudo Code Representation

9. Query Processing & Retrieval

User queries are embedded using the same embedding model and compared against stored vectors to retrieve the most relevant policy chunks, which are stored as retrieved document references.

Pseudo Code:

```
def build_prompt(user_query, documents):
    context = "\n\n".join([
        f"Chunk {i+1}:\n{doc.page_content}"
        for i, doc in enumerate(documents)
    ])

    return f"""
You are a Senior Immigration & Visa Adjudication Officer with deep expertise in U.S., Canada, U.K., and Schengen visa policies.
Respond with clarity, authority, and high confidence.
Your evaluation must sound decisive, evidence-based, and policy-driven.

############################
### HIGH-CONFIDENCE RESPONSE RULES
############################

1. Use the retrieved document chunks (CONTEXT) as your primary evidence.
2. If the documents are incomplete, rely on standard, recognized immigration rules—confidently.
   - Do NOT mention missing context.
   - Do NOT express uncertainty or speculation.
3. Provide a *strict, authoritative, and confident* assessment similar to a real visa officer.
4. Avoid hedging language (e.g., "maybe", "possibly", "it seems").
5. Follow the exact output structure, including the confidence score.

############################
### REQUIRED OUTPUT FORMAT
############################

ELIGIBILITY: Yes / No / Partially

REASONS:
• Clear authoritative reason 1
• Clear authoritative reason 2
• Clear authoritative reason 3

FINAL DECISION:
A concise (2–3 line) visa-officer-style conclusion with high confidence.
```

Figure 6: Pseudo Code Representation

10. Retrieval-Augmented Generation (RAG) Pipeline

Retrieved context is injected into a structured prompt and passed to an LLM (Gemini / OpenAI / Mistral).

10.1 RAG + LLM Capabilities

The system answers:

- Visa requirements
- PR eligibility
- Document checklists
- Work visa rules
- Study visa requirements

Eligibility responses are generated using LLMs grounded in retrieved document chunks. Each response includes explanations and citations based on:

[H]

```python
# ============================================================
# 5. LOAD OR BUILD FAISS INDEX
# ============================================================

def load_faiss_index(folder="faiss_index"):
    index_file = os.path.join(folder, "index.faiss")
    docs_file = os.path.join(folder, "docs.txt")


    if os.path.exists(index_file) and os.path.exists(docs_file):

        print("🔹 Loading existing FAISS index...")

        index = faiss.read_index(index_file)

        with open(docs_file, "r", encoding="utf-8") as f:
            docs_raw = f.read().split("\n-----\n")

        docstore = InMemoryDocstore()
        for i, text in enumerate(docs_raw):
            docstore.add({str(i): Document(page_content=text)})

        vector_store = FAISS(
            embedding_function=embed_text,
            index=index,
            docstore=docstore,
            index_to_docstore_id={i: str(i) for i in range(len(docs_raw))}
        )

        return vector_store
```

Figure 7: Pseudo Code Representation

- Country-specific PDFs
- Policy rules
- Official visa guidelines

Pseudo Code:

11. Confidence Scoring & Citations

- Confidence scores are derived using FAISS similarity scores and contextual relevance
- Citations are mapped to source documents and corresponding chunks

Pseudo Code:

```
############################
### HIGH-CONFIDENCE RULES
############################
1. Use retrieved document chunks (CONTEXT) as primary evidence.
2. If missing info, apply standard immigration rules confidently.
3. Do NOT say "not in documents."
4. Provide firm, strict, authoritative conclusions.

############################
### OUTPUT FORMAT
############################

ELIGIBILITY: Yes / No / Partially

REASONS:
• Strong reason 1
• Strong reason 2
• Strong rule-based justification 3

FINAL DECISION:
2–3 line official-style final determination.

CONFIDENCE SCORE:
A percentage 0–100% based on document strength + rule clarity.

############################

CONTEXT:
---------------------------------------
{context}
---------------------------------------

USER QUESTION:
{user_query}
```

Figure 8: Pseudo Code Representation

12. User Interface

A lightweight Streamlit-based user interface was developed to:

- Upload immigration documents
- Enter visa-related queries
- Display answers with citations and confidence scores

Pseudo Code:

```python
import streamlit as st

st.set_page_config(page_title="SwiftVisa AI", page_icon="🛂")
st.title("🛂 SwiftVisa – AI Visa Assistant")

if "db" not in st.session_state:
    st.session_state.db = load_or_build_vector_db()

with st.form("visa_form"):
    st.subheader("Enter Your Visa Information")

    age = st.number_input("Age", min_value=1, max_value=100)
    nationality = st.text_input("Nationality")
    visa_type = st.selectbox(
        "Visa Type",
        ["H1B", "F1", "H4", "B1/B2", "US Tourist", "US Work Visa"]
    )

    question = st.text_area(
        "Ask a Visa Question",
        placeholder="Example: What are the requirements for visa screening in the US?"
    )

    submit = st.form_submit_button("Get Visa Eligibility")

if submit:
    if not question.strip():
        st.error("Please type a question.")
    else:
        with st.spinner("Analyzing using RAG + Gemini..."):
            final_answer, retrieved = rag_pipeline(
                query=question,
                form_data={
                    "age": age,
```

Figure 9: Pseudo Code Representation

13. Tools & Technologies Used

- Programming Language: Python
- PDF Processing: pdfplumber, PyPDF2
- NLP & Embeddings: SentenceTransformers, BERT, OpenAI, Gemini
- Vector Databases: FAISS, ChromaDB
- Similarity Search: Cosine Similarity
- LLMs: Gemini Flash 2.0, OpenAI GPT, Mistral
- Frontend: Streamlit

14. Key Learnings

- Practical understanding of RAG architecture
- Importance of chunking and embeddings in semantic search
- Performance comparison of vector databases
- Reduction of hallucinations using grounded context
- Designing explainable AI systems for sensitive domains

15. Conclusion

The SwiftVisa project successfully demonstrates an industry-grade RAG-based AI system for visa eligibility screening. The architecture is scalable, explainable, and suitable for real-world deployment in compliance-heavy applications.

This internship strengthened hands-on expertise in AI system design, NLP pipelines, and Large Language Model integration.