Swift Visa AI-Based Visa Eligibility Screening Agent

Milestone 1: Create a Vector data base

1. Create Folder Structure for Country Visa Documents

I created a main folder named **"Visa_Eligibility"**, and inside this folder I added all the visa eligibility PDFs for the countries my agent currently supports. Right now, this folder contains **five PDFs**: USA, UK, Saudi Arabia, UAE, and Germany.

All these PDFs are placed directly inside the Visa_Eligibility folder so the pipeline can automatically find them and process them without any manual selection. This structure helps keep all the visa documents organized and makes it easy to add or update more country PDFs in the future.

2. Extract and Clean Text from PDF Files

Each PDF is processed to extract the text from all of its pages. The extracted text is merged into a single continuous block. Since visa PDFs often contain tables, headers, footers, symbols, and inconsistent formatting, the next step is to clean the text. This includes removing unnecessary characters, fixing spacing issues, converting text to lowercase, and removing unwanted symbols.
The cleaned version of the text is saved as a separate `.txt` file. The purpose of this process is to transform complex PDF structures into clean and uniform text that can later be chunked and embedded effectively.

3. Split Cleaned Text into Smaller Chunks

Once the cleaned text is ready, it is split into smaller pieces called chunks. These chunks are usually created based on a fixed number of words, such as 800 words per chunk. The text is broken into multiple segments, and each segment is saved as its own text file.
This step ensures that long documents are broken down into manageable units that can be embedded without exceeding the input limits of language models. Chunking also improves retrieval accuracy because each chunk represents a focused section, such as eligibility, documents required, fees, financial proof, interview rules, or processing time.

4. Convert Text Chunks into Embedding Vectors

Each chunk created from the previous step is passed into a Sentence Transformer model, such as **all-MiniLM-L6-v2**. The model converts each chunk into a numerical vector known as an embedding. These embeddings capture the semantic meaning of the text. For example, two chunks describing financial requirements, even if written differently, will have embeddings

placed close to each other.
All embeddings and their corresponding chunk texts are stored in memory during this step so that they can be added to a vector database later.

5. Store Embeddings in a FAISS Vector Database

All embedding vectors are collected and converted into a numerical array. A FAISS index is then created based on the size of these vectors. The embeddings are inserted into this FAISS index, which is then saved to disk as a `.faiss` file.
FAISS acts as a high-performance vector database, enabling fast similarity search. When the AI agent receives a user query, it converts the query into an embedding and compares it with the FAISS index to find the closest matching visa information. This results in quick and semantically accurate retrieval of relevant rules and requirements.

6. Automate the Pipeline to Process All PDFs in a Folder

Instead of running each step manually for every PDF, the pipeline is designed to process all files inside a given folder automatically. When executed, the system scans the folder, collects all PDF files, and runs them through the entire flow: extraction, cleaning, chunking, embedding, and FAISS storage.
This makes the pipeline scalable, allowing it to handle any number of visa PDFs—from a few to hundreds—without manual intervention. It becomes easy to add new documents or new countries by simply dropping PDFs into the correct folder.

7. Final Output of the Pipeline

After the pipeline finishes, you will have a complete set of outputs:

- Cleaned text files for every PDF
- Chunked text files that represent narrower sections of each visa document
- Embedding vectors for every chunk
- A FAISS index containing all embeddings
- A searchable knowledge base for visa information

The system is now ready for question-answering. The FAISS vector store enables the AI agent to instantly retrieve relevant information such as eligibility rules, application fees, document requirements, stay duration, restrictions, work rights, and financial proof requirements for any visa type across multiple countries.

Learnings

1. RAG –Retrieval Augmented Generation

Retrieval-Augmented Generation (RAG) is the process of optimizing the output of a large language model, so it references an authoritative knowledge base outside of its training data sources before generating a response. Large Language Models (LLMs) are trained on vast volumes of data and use billions of parameters to generate original output for tasks like answering questions, translating languages, and completing sentences. RAG extends the already powerful capabilities of LLMs to specific domains or an organization's internal knowledge base, all without the need to retrain the model. It is a cost-effective approach to improving LLM output so it remains relevant, accurate, and useful in various contexts.

In Simple words,

RAG is an AI technique where an LLM (like ChatGPT) is combined with an external knowledge base to give **accurate, up-to-date, factual answers**.

How RAG works

Without RAG, the LLM takes the user input and creates a response based on information it was trained on—or what it already knows. With RAG, an information retrieval component is introduced that utilizes the user input to first pull information from a new data source. The user query and the relevant information are both given to the LLM. The LLM uses the new knowledge and its training data to create better responses. The following sections provide an overview of the process.

Create external data

The new data outside of the LLM's original training data set is called *external data*. It can come from multiple data sources, such as a APIs, databases, or document repositories. The data may exist in various formats like files, database records, or long-form text. Another AI technique, called *embedding language models*, converts data into numerical representations and stores it in a vector database. This process creates a knowledge library that the generative AI models can understand.

Retrieve relevant information

The next step is to perform a relevancy search. The user query is converted to a vector representation and matched with the vector databases. For example, consider a smart chatbot that can answer human resource questions for an organization. If an employee searches, *"How much annual leave do I have?"* the system will retrieve annual leave policy documents alongside the individual employee's past leave record. These specific documents will be returned because they are highly-relevant to what the employee has input. The relevancy was calculated and established using mathematical vector calculations and representations.

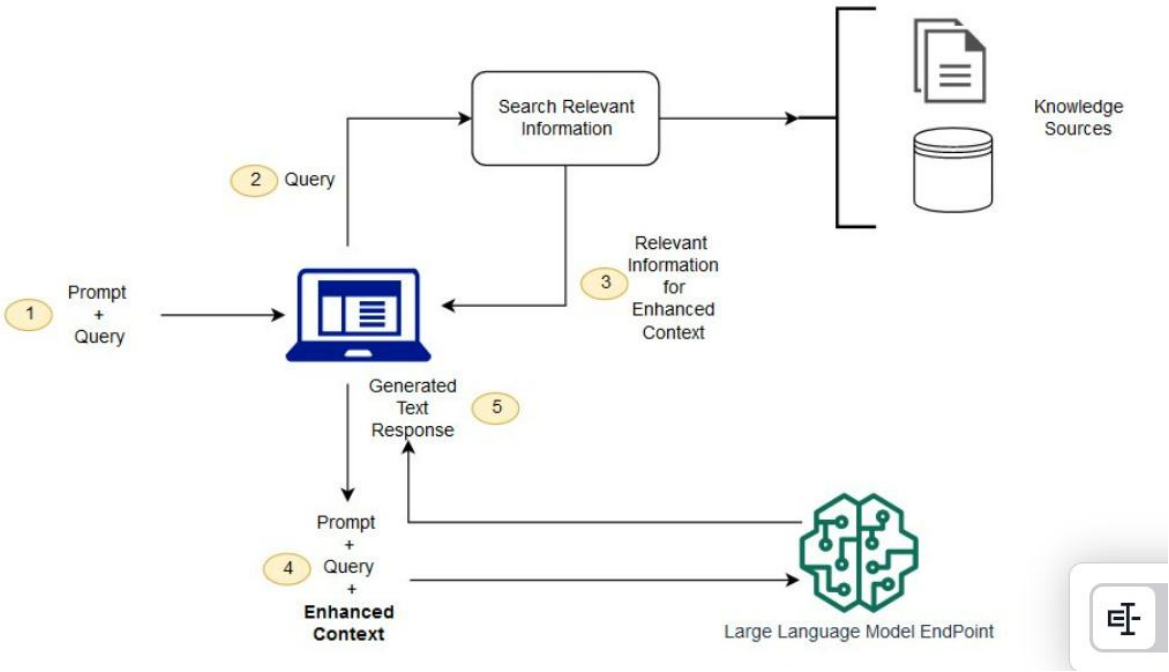
Augment the LLM prompt

Next, the RAG model augments the user input (or prompts) by adding the relevant retrieved data in context. This step uses prompt engineering techniques to communicate effectively with the LLM. The augmented prompt allows the large language models to generate an accurate answer to user queries.

Update external data

The next question may be—what if the external data becomes stale? To maintain current information for retrieval, asynchronously update the documents and update embedding representation of the documents. You can do this through automated real-time processes or periodic batch processing. This is a common challenge in data analytics—different data-science approaches to change management can be used.

The following diagram shows the conceptual flow of using RAG with LLMs



2. What are Embeddings?

Embeddings are a way to convert text such as words, sentences, or documents into numbers so that a computer can understand meaning. Humans understand ideas and context, but computers only understand numbers. So embeddings act like a bridge — they turn natural language into mathematical form.

Each piece of text is converted into a vector: a long sequence of numbers. For example, a single sentence might become a list of 384 numbers, or 768 numbers, depending on the model used. The important thing is not the individual numbers, but the pattern they form in vector space.

When two pieces of text mean similar things, their embeddings lie close to each other in this vector space. When they mean different things, their embeddings lie far apart. This is why embeddings allow computers to measure similarity in meaning, not just similarity in spelling or keywords.

Why embeddings are powerful

Traditional computer search worked through exact word matching. If I search "US Visa fee", traditional search engines would look for the exact words "visa", "fee", "US", and would not necessarily understand the meaning.

Embeddings solve this problem. They understand meaning and intent. The sentence “Cost of USA visa” and “How much is the American visa fee?” do not share many words, but mean almost the same thing. Embeddings capture that similarity. So a search system based on embeddings will put these two sentences close together in the vector space.

This is why modern AI search, RAG systems, intelligent chatbots, and semantic search engines rely on embeddings rather than keyword matching.

How embeddings are created

Embeddings come from machine learning models, usually based on transformers. These models have been trained on huge amounts of text. During this training, the models learn patterns of meaning, relationships between words, grammar, and even world knowledge. Because of this learning, the models can take a piece of text and output numbers that represent its meaning.

The embedding does not store the literal words. It captures semantic information such as:

- What topic the text belongs to
- The intent behind the sentence
- The relationship between ideas
- The emotional tone (in some models)
- The context implied by the sentence

So when a model produces embeddings, it is essentially placing each sentence at a specific location in its understanding of language.

Why embeddings are necessary for RAG

In a RAG system, users ask natural questions. The system must find the relevant information from the stored documents and give it to the LLM. But LLMs cannot search entire PDF files by themselves. Instead, we:

1. Split documents into chunks
2. Convert each chunk into embeddings
3. Store those embeddings in a vector database

When a user asks a question, we convert the question into an embedding and find which chunk is closest in meaning. That chunk is then given to the LLM as context, ensuring that the answer is fact-based rather than hallucinated.

Without embeddings, a RAG system cannot retrieve meaningful information from documents.

Embeddings do not understand spelling or surface-level patterns. They understand **meaning**.

For example, the words “study permit”, “student visa”, “permission to study abroad”, and “authorization for education in another country” are different phrases, but an embedding model will place them close to each other because the meaning is the same.

This ability to understand meaning is the whole reason embeddings revolutionized NLP.

How many numbers are in an embedding vector?

This depends on the model. Different models use different dimensions. Some common sizes:

- 384 dimensions
- 512 dimensions
- 768 dimensions
- 1,024 dimensions
- 3,072 dimensions (for some OpenAI models)

A larger embedding dimension does not always mean better quality. The key is how well the model has been trained to represent meaning.

Similarity between embeddings

Similarity in embeddings is calculated mathematically using vector similarity.
Because embeddings convert text into numerical vectors, we can measure how close two vectors are in a multi-dimensional space.
The closer the vectors, the more similar the meanings of the texts.

3. Different embedding generating modules

Embedding models (also called embedding generators) are AI models that convert text, images, or other data into numerical vectors so that meaning can be compared mathematically.

Below are the major categories and most popular modules/libraries used to generate embeddings.

3.1. Transformer-based Embedding Models

These are the most advanced and widely used models today for semantic embeddings.

Sentence Transformers (SBERT)

- Library: sentence-transformers
- Very popular for semantic similarity, RAG, Q&A, search
- Examples:
  - all-MiniLM-L6-v2
  - paraphrase-MiniLM-L12-v2
  - multi-qa-mpnet-base-dot-v1

BERT-family models

- BERT
- RoBERTa
- DistilBERT
- DeBERTa
- ALBERT
  (Sentence-BERT versions are optimized for similarity.)

Modern embedding models

- OpenAI Embeddings (text-embedding-3-small, text-embedding-3-large)
- Google Universal Sentence Encoder (USE)
- Cohere Embedding model
- NVIDIA Embedding model
- HuggingFace E5 models (e.g., intfloat/e5-large-v2)
- LLM-based embedding models (LLaMA, Mistral, etc. for feature extraction)

3.2. Word-level Embedding Models

These generate vectors per word**,** not complete sentence meaning. They are older but foundational.

- Word2Vec
- GloVe (Global Vectors)
- FastText
- ELMo (Embeddings from Language Models)

Limitations:

- No sentence-level meaning
- Does not understand context differences (e.g., Apple fruit vs Apple company)

3.3. Character- / Subword-based Embedding Models

These embed smaller units such as characters or subwords.
Useful for languages with rich morphology or spelling variations.

Examples:

- FastText (word + character n-grams)
- Byte Pair Encoding (BPE) embeddings
- GPT token embeddings

4. Importance of using FAISS or chroma dB

When working with embeddings, the key problem is retrieving the most similar vectors quickly and accurately from a large collection.

Embeddings convert text into high-dimensional numerical vectors.
A real-world system quickly grows from hundreds to thousands, then millions of vectors.

Performing similarity comparison manually — checking a query embedding against every embedding in the dataset — is extremely slow for large data sizes, because:

1. Each embedding may be 384, 768, 1024 or 1536 dimensions
2. The system must compute similarity for every document chunk
3. This has to happen for every user query

Even on a fast CPU, physically comparing a query vector to millions of stored vectors would cause:

- Slow responses (seconds or minutes)
- Laggy user experience
- High compute cost
- Inability to scale

So embeddings alone are not enough**.**
You need a specialized search engine / database optimized for vectors**.**

That is exactly where FAISS and ChromaDB come in.

How FAISS or ChromaDB solve the fundamental bottleneck

What companies discovered (Meta, OpenAI, Google, Anthropic, Amazon etc.) is:

Most vectors in a huge knowledge base are not relevant to a query.

So instead of comparing the query with every vector, they developed mathematical indexing structures that allow the system to:

- Prune 99.9% of irrelevant vectors instantly
- Search only inside the most promising clusters
- Return results in milliseconds, not seconds

FAISS and ChromaDB do that through:

- Advanced vector indexing algorithms
- Approximate nearest neighbour (ANN) search

- Quantization + clustering techniques
- Memory-optimized layouts
- Parallel / GPU processing

This transforms embedding-based search from slow → real-time**.**

FAISS

FAISS (Facebook AI Similarity Search) is a high-performance vector similarity search engine**.**
It is not a database — it is an algorithmic engine for fast vector search**.**

Its importance lies in:

- The ability to store and search millions to billions of embeddings
- Custom indexing algorithms designed for high-dimensional spaces
- Extremely fast inference on CPU and GPU
- Support for memory-compressed indexing (so vectors don't consume massive RAM)
- High recall (it finds the closest matches very accurately)

FAISS became the industry standard when the need emerged for:

- Instant retrieval
- Large-scale document search
- Recommendation systems
- AI agents
- RAG

FAISS makes semantic search fast and scalable. Without it, embeddings alone are not usable in large datasets.

ChromaDB

ChromaDB is a vector database**,** which means it goes beyond just search.

Its importance comes from providing a complete data management layer around embeddings**,** including:

- Storing documents + embeddings + metadata
- Persistent storage (data survives restarts)
- Fast vector lookup
- Metadata filtering (e.g., only India Visa documents, or date range)
- Integration with LLM frameworks (LangChain, LlamaIndex, etc.)

ChromaDB focuses on:

- Ease of use
- Production readiness
- Retrieval + data organization
- Developer-friendly APIs

What makes ChromaDB powerful is that it acts like:

A memory system for LLMs

Instead of recomputing embeddings every time, you store them in ChromaDB and just query them.

RAG pipeline without FAISS / ChromaDB

Imagine a chatbot that must answer questions based on 50,000 PDF pages.

Without a vector search engine:

- Every query requires looping over all documents
- Every loop requires computing similarity
- Each similarity requires 384–1536 dimensional vector math

The result:

- High latency
- High compute usage
- Unusable in production

Even if embeddings are correct, retrieval becomes the bottleneck.
A fast LLM is worthless if *retrieval is slow*.

What happens with FAISS or ChromaDB

Query flow becomes extremely efficient:

1. User asks a question → embed the question
2. Vector database computes similarity search in milliseconds
3. Only the top-k most relevant document chunks are retrieved
4. LLM receives focused, relevant context
5. Response becomes faster and more accurate

This leads to:

- Very low hallucination rate (because relevant info is retrieved)
- Very fast responses
- Reduced prompt sizes → lower cost
- Improved user satisfaction