Project Name: SwiftVisa AI-Based Visa Eligibility Screening Agent

Project summary (one line)

SwiftVisa is a retrieval-augmented screening agent that indexes official visa-policy documents per country into a vector database (FAISS) so the agent can quickly retrieve relevant policy chunks for eligibility-checking or question answering.

Milestone 2: Build the RAG Pipeline and User Interfaces

Goal

Implement the complete Retrieval-Augmented Generation (RAG) pipeline to enable grounded question answering, incorporate confidence scoring, and deploy user-friendly interfaces (CLI and Streamlit) for system interaction and evaluation.

List of steps to achieve the goal:

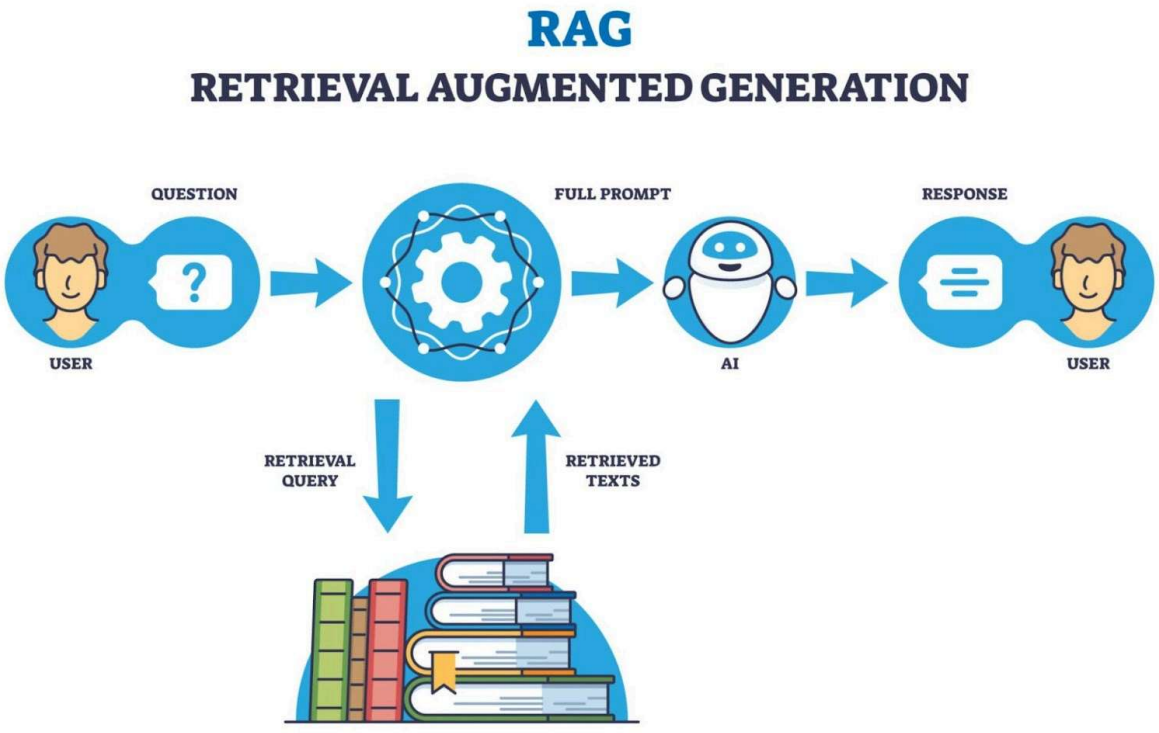
1. **Retrieval Logic:** Implement the query embedding and FAISS similarity search.
2. **Prompt Engineering:** Design the system prompt and user context template for grounded generation.
3. **LLM Integration:** Integrate the Gemini/OpenAI client for structured JSON output.
4. **Confidence Scoring:** Develop a method to calculate and report a blended confidence score.
5. **Interfaces:** Build the CLI and Streamlit dashboard.
6. **Evaluation:** Implement batch query processing for testing and performance analysis.

Implementation Steps:

1. **Retriever (rag/retriever.py):**
   - Accepts a user query.
   - Embeds the query using the same SentenceTransformer model used for indexing.
   - Performs a Nearest Neighbor (kNN) search against the persisted FAISS index to retrieve the top K relevant document chunk IDs and their similarity scores.
   - Maps the IDs back to the original chunk text and metadata using the persisted JSON file.
2. **Prompt Builder (rag/prompt_builder.py):**
   - Constructs a comprehensive system instruction that enforces the **Analyst Persona** and demands **structured JSON output** for eligibility analysis.
   - Creates the final prompt by injecting the user's original query, and the retrieved

document chunks, framing the chunks as the "only source of truth."

3. **LLM Client (rag/llm_client.py):**
    - Wraps the Gemini API calls.
    - Enforces the output schema using the LLM's **Structured Output** feature (JSON schema).
    - Implements a retry mechanism and robust try/catch logic to handle common API errors and invalid JSON responses.
4. **Pipeline (rag/pipeline.py):**
    - Acts as the orchestrator, chaining the retrieval and generation components.
    - Calculates the **Blended Confidence Score** (e.g., combining the max retrieval score with the LLM's self-reported confidence).
    - Logs the query, retrieved chunks, and final answer using the rag/logger.py.
5. **User Interfaces:**
    - **CLI (query_cli.py):** Provides a simple, interactive console interface for immediate testing.
    - **Dashboard (streamlit_app.py):** Creates a responsive web interface for easier RAG testing and visual display of the structured response and retrieval metadata.



Getty Images

Explore

Deliverables produced (code files)

Files created or updated in this milestone:

File Name	Purpose and Function
rag/retriever.py	Implements the query embedding and FAISS search to get relevant chunks and scores.
rag/prompt_builder.py	Responsible for generating the strict system prompt and injecting RAG context.
rag/llm_client.py	Wrapper for LLM APIs, handles JSON schema enforcement and error resilience.
rag/pipeline.py	The main RAG execution logic; orchestrates retrieval, generation, and confidence scoring.
rag/logger.py	Captures queries, retrieved content, and final structured output for logging/auditing.
query_cli.py	Command-line interface allowing users to input profile data and queries.
streamlit_app.py	Streamlit Dashboard for an enhanced, visual user experience (Live Test Console).
process_test_queries.py	Script to run batch queries from user_queries.json and store structured results in query_results.json.

Learnings from Milestone 2:

1. **Structured LLM Generation:**
    - Understanding the importance of enforcing a JSON schema in the prompt/API call to ensure predictable, machine-readable output (decision, reason, confidence, future_steps). This is critical for reliable downstream processing and display.
2. **RAG Efficacy and Prompting:**
    - Mastering the RAG prompt template to instruct the LLM to prioritize the *provided context* (retrieved chunks) over its general knowledge, thereby guaranteeing a

grounded response.

3. **Confidence Metrics:**
    - Implementing a combined confidence score—blending the quantitative FAISS retrieval score (how relevant the documents are) with the LLM's qualitative assessment (how confident it is in the final answer)—to provide a robust reliability metric to the user.
4. **Deployment and Interface:**
    - Gaining experience in transitioning a core Python script into user-facing applications (CLI for developers, Streamlit for business users).

Common pitfalls & how I handled them

Problem	Symptom	Fix implemented
LLM Hallucination	LLM ignores retrieved facts and relies on outdated internal knowledge.	Implemented a **strict system instruction** demanding the LLM reference the provided context *only* and to state "I cannot answer this based on the provided documents" if context is insufficient.
JSON Parsing Failure	LLM outputs text instead of the required JSON structure, breaking the pipeline.	Used the LLM API's native **Structured Output** feature to enforce the JSON schema. Added fallback Python try-catch and up to 3 retries with a corrective meta-prompt if parsing failed.
Irrelevant Retrieval	FAISS returns chunks with a high score that are semantically irrelevant to the *specific* question.	Reduced the maximum chunk size during indexing (Milestone 1) and fine-tuned the retrieval K value in rag/retriever.py to balance precision and recall. Incorporated the blended confidence score to flag low-quality answers.

API Rate Limits	Scripts failed during batch testing due to exceeding API request limits.	Implemented a robust **exponential backoff** strategy in rag/llm_client.py to automatically wait and retry failed API calls, ensuring high availability during large batch runs.