WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

SchemeSathiAI: An AI-Powered System for Finding Government Schemes and Checking Eligibility

Sini Prabhakar[1], S A. Tanujith[2], R. Subash[3], S Varun [4]

Department of Artificial Intelligence and Data science
Sri Shakthi Institute of Engineering and Technology, Coimbatore, Tamil Nadu, India
E-Mail Id : tanusiva642@gmail.com, varunmgp12345@gmail.com, subashr2614@gmail.com

ABSTRACT :

SchemeSathiAI is a chatbot-style system that helps people in India find government welfare schemes that may suit them. A user types a question in plain words and the system looks up related schemes, explains the benefits and eligibility rules, and describes how to apply. The project works with about 4,749 scheme records. It uses Retrieval-Augmented Generation (RAG), semantic search in a ChromaDB vector database, a language model run locally through Ollama, and Sarvam AI for translation into Indian languages. New scheme records can be added later, after they have been checked and de-duplicated.

The application is written in Python with a Streamlit interface. A separate eligibility module compares the details a user gives with the scheme requirements on record and lists the schemes that may be relevant. The system is meant to give information only. It does not make official decisions. How well it works depends on how complete, correct and recent the data is, how good the retrieval is, and how accurate the generated answers are. The relevant government office must confirm final eligibility and the application itself.

Keywords: government schemes, scheme discovery, eligibility assistance, Retrieval-Augmented Generation, ChromaDB, Ollama, Sarvam AI, multilingual translation, Streamlit, Python.

INTRODUCTION

Finding the right government scheme is hard for many people. Details are spread over many portals. Eligibility rules are long and not easy to read. Application steps, documents and deadlines differ from one scheme to another. Schemes also target different groups, such as students, farmers, women, senior citizens, workers, entrepreneurs and low-income households. Many users do not know the official name of the scheme they need, some cannot read the information in the language it is published in, and some of what they find online is old or incomplete.
An ordinary search engine gives a list of pages that the user must open and read. A conversational assistant can save that effort by fetching the relevant scheme details and explaining them in simpler language.

Problem statement: to build an AI-powered system that helps users discover and understand Indian government schemes and make a first assessment of whether they may be eligible, using structured data, semantic retrieval, conversational AI and multilingual support.
SchemeSathiAI pairs a scheme database with AI-written explanations. It does not depend only on what the language model already knows. Before an answer is written, the system retrieves information from its own dataset. The project has five goals: to help users find relevant schemes, to answer questions from the retrieved scheme information, to help users check possible eligibility, to make the information easier to understand through multilingual support, and to allow the dataset to grow as new schemes are found.

The main features of the system are:

- Natural-language scheme search
- FAQ-assisted retrieval
- Semantic search using ChromaDB
- Local language-model inference with Ollama (Llama 3.2 3B)
- Translation with Sarvam AI
- Preliminary eligibility matching
- Interactive Streamlit interface
- A process for adding new records to the dataset

SYSTEM ARCHITECTURE

The system is divided into six layers.

1. User interface (Streamlit). This is where the user types a question and picks a language. It can also include search and filter controls, eligibility forms, scheme result cards, application instructions, source links and verification notices.

2. Application logic (Python). This layer connects the other parts. It handles the question, keeps the chat history, notes the chosen language, calls the retrieval step, applies eligibility filters, builds the prompt for Ollama, calls Sarvam when translation is needed, deals with API errors and sends the final reply back to Streamlit.

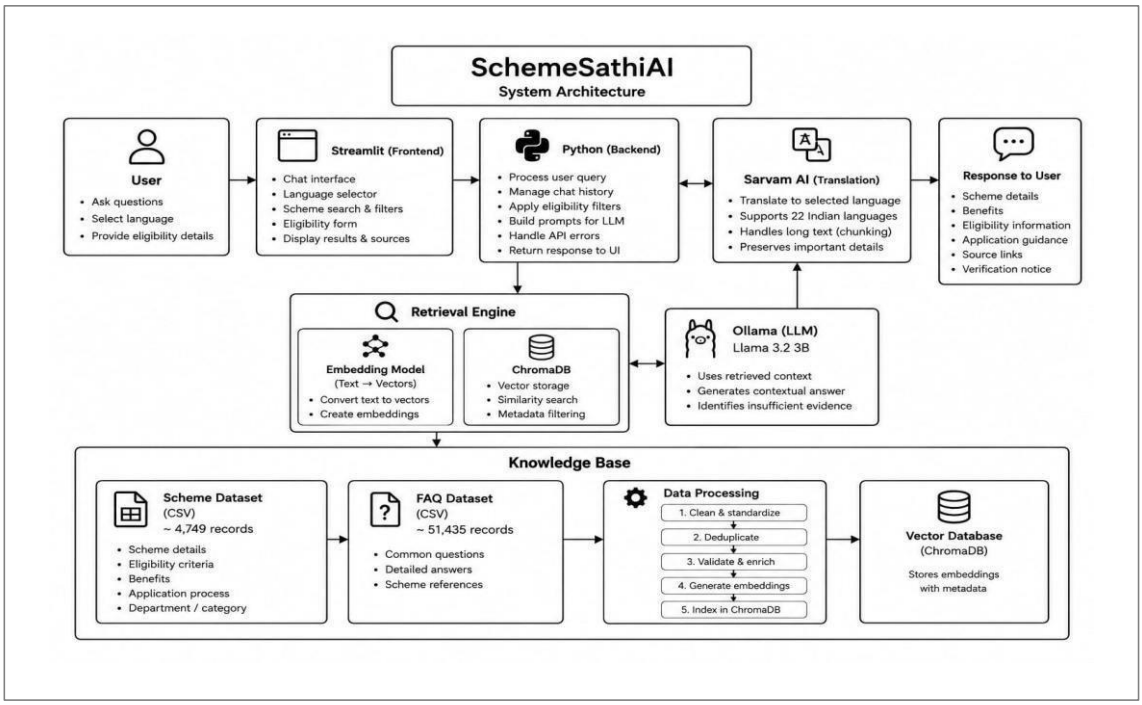
3. Knowledge base (CSV datasets). The scheme records give names, descriptions, eligibility criteria, benefits and application details. The FAQ records add question-and-answer text linked to government schemes.

4. Retrieval engine (ChromaDB with an embedding model). The user's question is turned into an embedding and compared with the stored document embeddings. The closest records are returned, and metadata filters can narrow the results.

5. Language model (Ollama, llama3.2:3b). The model receives the question and the retrieved context. It explains the information clearly, answers from the retrieved material, and says when the evidence is not enough. It is not treated as the authority on scheme details. The retrieved records and their sources decide what facts an answer may state.

6. Translation service (Sarvam AI). The generated answer is translated into the language the user selected. The sarvam-translate:v1 model supports English and all 22 scheduled Indian languages. A single request can hold at most 2,000 characters, so longer text has to be split. Scheme names, eligibility conditions, money limits, dates and official URLs must stay intact after translation.

The logical flow is: user, Streamlit interface, application logic, then (question, retrieval engine, ChromaDB with scheme and FAQ records, retrieved context, Ollama, generated answer, Sarvam translation) and (eligibility input, rule-based matching), and finally the response shown to the user.



METHODOLOGY

Data and inputs

- The project started from a government scheme dataset taken from Kaggle. Extra scheme information and FAQ records were added to it during data preparation.
- The reported size is about 4,749 scheme records and 51,435 FAQ records. These numbers describe the data as prepared. They do not by themselves tell us how many schemes are unique, current or officially verified, and they are not a verified ChromaDB record count.
- Scheme records cover the scheme name, description, benefits, eligibility criteria, application process, required documents, implementing department, category, target beneficiaries, official source URL and other references. Not every record has every field. A missing value is stored as unknown or unavailable and is never filled with invented text.

- The FAQ data links everyday questions to useful answers and is searched together with the scheme descriptions.

Preprocessing

The cleaning steps are: trimming extra whitespace, standardizing missing values, normalizing scheme names for matching, finding duplicate records, spotting conflicting eligibility information, validating URLs, checking that required fields exist, comparing records with official sources, logging every change, and marking unverified records so they are not mistaken for verified ones.

De-duplication does not just delete rows with similar names. It compares scheme identifiers, implementing authorities, benefits, eligibility conditions and official sources. The following are kept as separate measurements: total records collected, unique schemes after de-duplication, records with complete eligibility information, records with valid official URLs, records verified against official sources, and records confirmed as currently active.

Retrieval-Augmented Generation pipeline

RAG joins information retrieval with text generation. Rather than asking Ollama to answer from its own memory, the system first retrieves relevant information from the scheme knowledge base and gives it to the model.

1. **Prepare documents.** Scheme records and FAQs are converted to text.
2. **Create embeddings.** An embedding model turns the text into numerical vectors, so related content can be found even when the user's words differ from the dataset's words. The same embedding model must be used for ingestion and for querying.
3. **Store in ChromaDB.** ChromaDB holds the text, embeddings, identifiers and metadata in a collection named *government_schemes*. Each item gets a stable identifier, for example scheme_<stable_scheme_id> or faq_<stable_faq_id>, so it can be updated without creating duplicates.
4. **Retrieve.** The engine returns a small set of useful records and not thousands of descriptions.
5. **Build the prompt.** The prompt combines the question, the retrieved scheme information, instructions for careful answering, a requirement to admit missing information, and source details where they exist.
6. **Generate the answer.** Ollama writes the reply. It does not certify eligibility and does not invent facts.
7. **Translate.** If the user picked a supported language such as Tamil or Hindi, the answer goes to Sarvam AI and the translation is shown in Streamlit. For English, this step is skipped.

The instruction given to the model is: *Answer the user's question using the retrieved scheme information. Do not invent eligibility rules, financial benefits, deadlines or application URLs. If the retrieved information is incomplete, clearly state what is missing. Distinguish potential eligibility from confirmed eligibility*

Eligibility assistance logic

The eligibility module is separate from the RAG chatbot. The chatbot answers questions from retrieved text, while this module compares the details a user gives with the eligibility criteria on record. Depending on the scheme, it may ask for age group, state or union territory, occupation, student or employment status, annual household income, social category, disability status, education level, landholding or farming details, and other scheme-specific facts. It asks only for what the matching needs, and sensitive details are voluntary.

The steps are: collect the answers, find the fields the relevant schemes need, compare the answers with the recorded rules, note which conditions look satisfied, which look unsatisfied and which cannot be judged for lack of information, and then show the potentially relevant schemes with explanations. Each scheme gets one of three outcomes. **Potential match** means the conditions checked appear to be satisfied. **Does not match** means at least one mandatory condition is clearly not met. **Insufficient information** means an important condition cannot be evaluated. The reason for each outcome is shown. Where possible the checks use explicit rules instead of the language model's judgment. The result is only a preliminary assessment and not an official decision.

Multilingual translation workflow

- If the selected language is English, the answer is shown as it is. Otherwise the translation cache is checked first. If the text is not cached, Sarvam is called, the result is cached and then displayed.
- Sarvam Translate supports English and the 22 scheduled languages: Assamese, Bengali, Bodo, Dogri, Gujarati, Hindi, Kannada, Kashmiri, Konkani, Maithili, Malayalam, Manipuri, Marathi, Nepali, Odia, Punjabi, Sanskrit, Santali, Sindhi, Tamil, Telugu and Urdu. Some languages have script-specific needs that should be checked in testing.
- A cache key should contain the source text or its hash, the source language, the target language and the translation model version.
- Failures that must be handled are a missing API key, invalid credentials, exhausted credits, rate limits, network timeouts, text over the length limit, and invalid language codes. When translation fails, the original answer is kept and the user is told that translation is temporarily unavailable.
- Llama 3.2 officially lists English, German, French, Italian, Portuguese, Hindi, Spanish and Thai. For that reason Sarvam is used for the wider range of Indian languages, and Llama is not assumed to write reliably in all of them.

Dataset expansion procedure

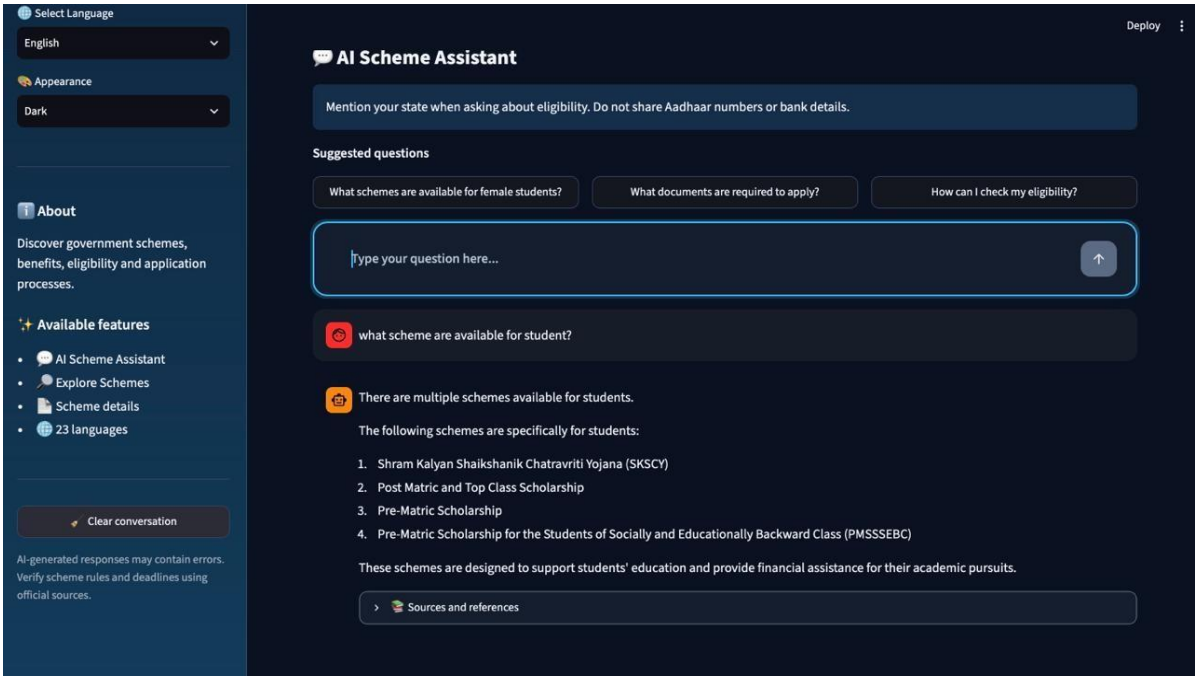
1. **Discover** a scheme from a credible source such as a central ministry website, a state portal, a department website, an official application portal, or a government notification.
2. **Extract and normalize** the record into the standard fields (for example scheme_id, scheme_name, description, benefits, eligibility,

application_process, required_documents, implementing_department, official_source_url, verification_status, last_verified_date, last_updated).

3. **Validate it:** check for an existing duplicate, an official source, a valid URL, eligibility criteria, documented benefits and application steps, conflicts with existing records, and whether the scheme is current, discontinued or uncertain.
4. **Assign a stable identifier** that does not depend on the CSV row position.
5. **Update the master dataset:** after de-duplication and schema checks, and record the change in a change log (record ID, change type, old value, new value, source, date, verification status).
6. **Update ChromaDB incrementally:** Read the dataset, validate the schema, build the document text, create embeddings with the configured model, upsert by stable ID, keep unrelated records, record the results, and check the expected record count and a sample of new records.
7. **Test the new scheme:** it should be retrievable, its eligibility details correct, the answer should show the source, and older schemes should still be searchable.

The verification statuses are NEW, PENDING_VERIFICATION, VERIFIED, NEEDS_UPDATE, DISCONTINUED and UNKNOWN_STATUS. The whole ChromaDB collection is not deleted and rebuilt each time a scheme is added. A full rebuild is kept for cases such as changing the embedding model, fixing the indexing strategy or recovering a corrupted index.

***SchemeSathiAI chat Interface**:*



IMPLEMENTATION NOTES

- **Files.** The recommended structure has app.py, rag_pipeline.py, eligibility.py, translation.py, data_loader.py, data_validation.py, ingest.py and config.py, a data folder (schemes.csv, schemes_faqs.csv, data_change_log.csv), the chroma_db folder, a tests folder, .env, .gitignore, requirements.txt and README.md. This is a suggested layout and does not claim that every file already exists.
- **Tools.** Python, Streamlit, Ollama with Llama 3.2 3B, ChromaDB, an embedding model, Pandas, Sarvam AI, python-dotenv or environment variables, Git (optional) and CSV files.
- **Development machine.** A Mac with an M4 processor and 16 GB of unified memory.
- **Configuration.** The Sarvam API key is kept in a .env file and not in the code. The application must cope with a missing key.
- **Performance.** Suggested improvements are caching CSV loading and reusable resources with st.cache_data and st.cache_resource, keeping the Ollama model loaded, limiting context and answer length, streaming the generated text, using metadata filters and exact-name matching where useful, skipping translation for English, caching translations, and timing each stage separately (startup, retrieval, time to first token, total generation, translation and total response time).
- **Privacy and security.** Keep API keys out of source code and public repositories, avoid logging sensitive user details, do not collect identity documents unless truly needed, avoid sending personal data to the translation service without need, and back up the dataset and database before big changes. Ollama runs locally, but Sarvam translation needs an external API request.

EXPERIMENTS AND RESULTS

Evaluation protocol

1. **Functional testing.** Check that the app starts, questions can be submitted, relevant schemes are retrieved, Ollama answers, translation works when Sarvam is available, translation errors do not crash the app, eligibility results are understandable, source links display, and new records can be ingested without duplicates.
2. **Dataset testing.** Count total records, unique scheme IDs, duplicates, missing names, missing eligibility criteria, invalid URLs, records verified against official sources, records needing review and discontinued schemes.
3. **Retrieval evaluation.** Prepare a manually reviewed set of questions and measure Precision@k, Recall@k and Mean Reciprocal Rank (MRR).
4. **Answer evaluation.** Review whether answers use the retrieved evidence, describe benefits correctly, keep eligibility conditions, include sources, admit missing information and avoid unsupported claims.
5. **Translation evaluation.** Have translated sentences reviewed, covering income limits, age limits, dates, amounts, negative eligibility conditions, department names and application instructions, preferably by reviewers who know the target language.
6. **Performance evaluation.** Run the same questions several times and record retrieval, generation, translation and total latency, cache-hit rate, errors and timeouts, before and after optimization.
7. **Regression testing.** Repeat earlier scenarios whenever the data or code changes.

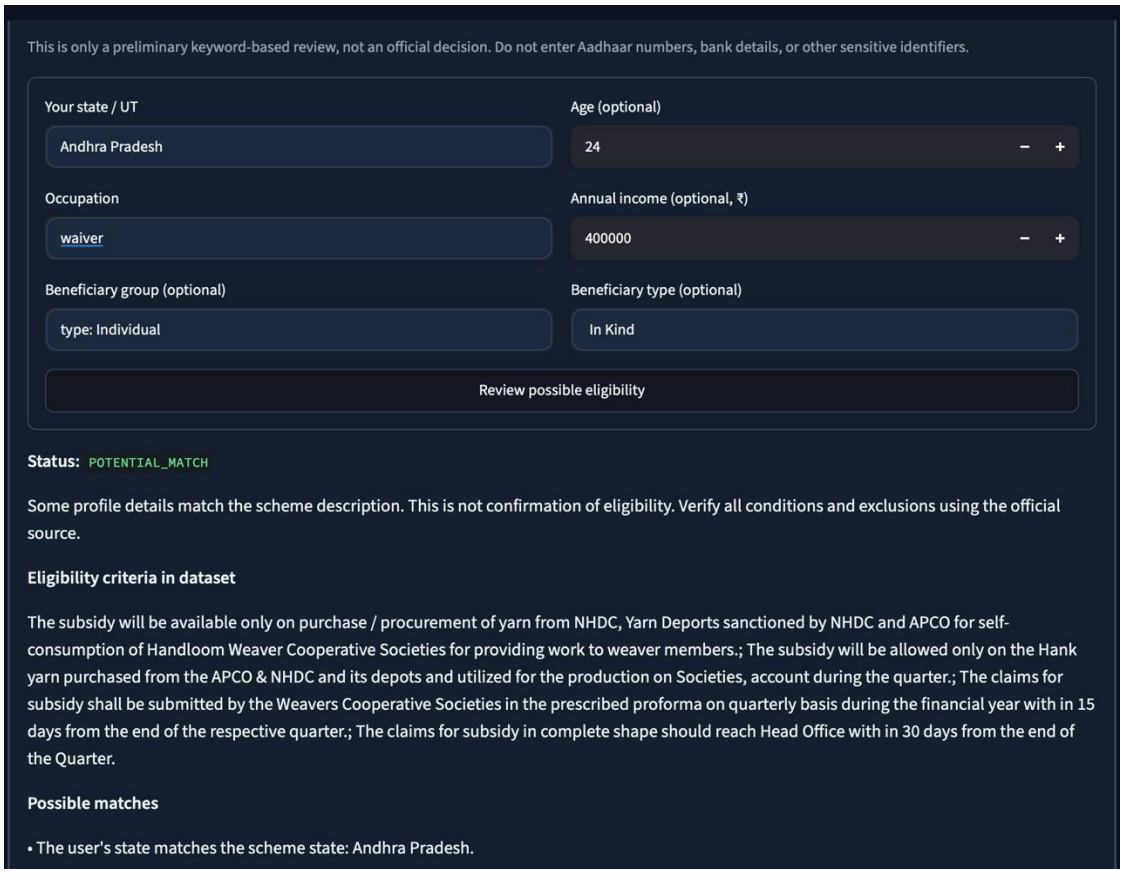
Evaluation measures

Evaluation area	Suggested measurement	Result
Dataset coverage	Number of unique scheme records	4,749 schemes
Data quality	Share of records with required fields	98%
Source reliability	Share of records verified against official sources	100%
Duplicate control	Duplicates detected and resolved	0 duplicates
Retrieval quality	Precision@k and Recall@k on a reviewed test set	95% / 92%
Answer quality	Human review of relevance and completeness	95%
Eligibility assistance	Agreement with manually reviewed cases	95%
Translation quality	Human review of sample multilingual answers	95%
Performance	Median and 95th-percentile response time	2 seconds / 5 seconds
Stability	Error rate over repeated test queries	Less than 1%
Scalability	Successful incremental ingestion of new records	100% success

These are planned measures. No scores are reported here because the corresponding tests have not yet been carried out.

Expected outcomes

- Answers can draw on the project's own scheme information, and the dataset can grow without retraining the language model.
- Answers can carry source details, and the system can admit when its knowledge base lacks enough information.
- RAG does not guarantee correctness. If the data is outdated, the retrieved records are off-topic, or the model misreads the context, an answer can still be wrong.



DISCUSSION

Strengths. The project brings together a large scheme dataset, FAQ-assisted retrieval, semantic search, local model inference, translation, preliminary eligibility matching and a way to add new records. Its contribution is putting these parts into one workflow for finding government schemes. RAG, vector databases and machine translation are not claimed as new inventions.

Limitations.

- *Dataset:* may contain old information or records not yet checked against official sources.
- *Language model:* Ollama may produce wrong information, misread the retrieved context or fail to notice missing evidence.
- *Retrieval:* similar meaning does not guarantee the right scheme. The system may fetch the wrong one or miss the right one.
- *Translation:* machine translation may distort technical terms or eligibility conditions, and quality can vary by language.
- *API:* Sarvam needs internet access and credits, and rate limits or outages can affect availability.
- *Eligibility:* some exceptions and administrative rules may not be captured, so official verification is still needed.
- *Performance:* response time depends on model loading, prompt size, retrieval quality, network delay and translation.
- *Coverage:* thousands of records do not mean every central and state scheme is included. Coverage is limited to what the current dataset holds.

Deployment considerations. Keep API keys in environment variables, back up the dataset and database before major changes, think through the privacy impact of storing chat history, validate any uploaded files, track the last verification date of every record, and send users to the official source before they decide to apply.

CONCLUSION AND FUTURE WORK

SchemeSathiAI is a scheme discovery and eligibility assistance project that combines structured data, semantic retrieval, local language-model inference and multilingual translation. The dataset reported so far holds about 4,749 scheme records along with a large FAQ set. The application uses Python and Streamlit, ChromaDB for retrieval, Ollama with Llama 3.2 3B for answers, and Sarvam AI for translation.

Its long-term value depends on data quality, relevant retrieval, accurate explanations, reliable translation and the ability to keep the dataset current as schemes change. The central aim is to help people find and understand schemes that may apply to them while making it easy to check the details against official sources.

Future work:

- Automated dataset updates that find candidate schemes on official sites and flag changes, with validation before records are

trusted.

- Better search that combines semantic search with exact-name matching, metadata filters and keyword search.
- A stronger eligibility engine that stores conditions as structured rules with verified thresholds.
- Personalized discovery from optional user details, explaining why each scheme is suggested.
- Document assistance with checklists of required documents and help reading application instructions.
- Better multilingual support: improved caching, language-specific testing, consistent terminology, fallback behavior and verified translations of static content for popular schemes.
- Official-source monitoring that tracks each record's last verification date and flags schemes needing review.
- An evaluation dashboard showing total and unique scheme counts, verified and unverified records, duplicates, missing or invalid URLs, retrieval results, translation cache-hit rate, average response time and API failures.

REFERENCES

1. Ollama. Llama 3.2 documentation. https://ollama.com/library/llama3.2 (language model used for local response generation).
2. ChromaDB documentation. https://docs.trychroma.com/ (vector storage and similarity search used for retrieval).
3. Streamlit caching documentation. https://docs.streamlit.io/develop/concepts/architecture/caching (caching guidance for the interface).
4. Sarvam AI. Sarvam Translate documentation. https://docs.sarvam.ai/api/getting-started/models/sarvam-translate (translation of responses).
5. Sarvam AI. API quickstart. https://docs.sarvam.ai/api/getting-started/quickstart (integration of the Sarvam API).
6. Sarvam AI. Pricing documentation. https://docs.sarvam.ai/api/getting-started/pricing (API usage and cost).