Technical Report: ESILV Smart Assistant (GenAI Project)

Date: January 4, 2026
Authors: Raphael Roux, Adrien Servas
Project: ESILV Agentic RAG Chatbot

Table des matières

1. Abstract

This report details the design, implementation, and evaluation of the **ESILV Smart Assistant**, a Generative AI application designed to serve as an intelligent interface for the ESILV engineering school. The system leverages **Retrieval-Augmented Generation (RAG)** to answer queries based on official documentation and employs **Agentic workflows** to handle interactive user registration.

Built on a modern stack comprising **LangGraph**, **Google Gemini**, **ChromaDB**, and **Streamlit**, the project demonstrates a scalable architecture for specialized domain assistants.

---

2. Problem Description

2.1. Context

Educational institutions like ESILV receive a high volume of repetitive inquiries from prospective students, current students, and parents. These inquiries often cover:

- Academic programs and majors.
- Admission requirements and deadlines.
- Campus life and extracurriculars.

Simultaneously, the administration aims to capture lead information (prospective students) efficiently without requiring them to fill out static forms, which often have lower conversion rates than conversational interfaces.

2.2. Objectives

The project aims to solve these challenges by building a system that:

1. **Automates Information Retrieval**: Instantly answers questions using a distinct knowledge base (brochures, PDFs), reducing administrative workload.
2. **Conversational Registration**: "Smartly" collects user information (Name, Email, Interest) during the chat, providing a seamless user experience.
3. **Contextual Memory**: Remembers previous interactions to provide coherent, multi-turn conversations.

---

3. Architecture

The system follows a modular, client-server-style architecture (though currently monolithic in deployment) with distinct layers for the User Interface, Logic/Orchestration, and Data Storage.

3.1. High-Level Overview

The application utilizes a **State-Based Agent** architecture. Unlike a simple linear chain (Prompt -> LLM -> Output), the agent operates as a graph of nodes (Chatbot and Tools) that loop until a satisfying response is generated or user input is required.

```
graph TD
    User[User] <--> UI[Streamlit UI]
    UI <--> Agent[LangGraph Agent]

    subgraph "Agent Logic (agents/graph.py)"
        State[State (Messages)]
        ChatNode[Chatbot Node]
        ToolNode[Tool Node]

        ChatNode -- Call Tool --> ToolNode
        ToolNode -- Result --> ChatNode
        ChatNode -- Final Answer --> UI
    end

    subgraph "Capabilities"
        RAG[RAG Tool] --> VectorDB[(ChromaDB)]
        Reg[Registration Tool] --> JSON[(registrations.json)]
    end

    subgraph "External Services"
        LLM[Google Gemini 2.5 Flash]
        Embed[Google GenAI Embeddings]
    end

    Agent -.-> LLM
    VectorDB -.-> Embed
```

3.2. File & Component Breakdown

- **User Interface (ui/app.py & app/cli.py):**
    - **Web App**: Built with Streamlit, providing a Chat tab for interaction, an Upload tab for knowledge base management, and an Admin tab for viewing registrations. It manages session state (history) and streams events from the agent graph.
    - **CLI**: A terminal-based implementation for headless testing and quick interaction loops.
- **Orchestration (agents/graph.py):**
    - **LangGraph**: Defines the control flow. The graph consists of a

chatbot node (calls the LLM) and a tools node (executes Python functions).

    - **Memory**: Uses MemorySaver to persist conversation threads, allowing the LLM to recall past questions within a session.
    - **Prompt Engineering**: A system prompt defines the persona ("ESILV Smart Assistant") and instructs the model on when to use specific tools versus general conversation.
- **Data Ingestion (ingestion/ingest.py):**
    - Handles the ETL (Extract, Transform, Load) process.
    - **Extract**: Reads PDFs using PyPDFLoader.
    - **Transform**: Splits text into 1000-character chunks with 200-character overlap to preserve context at boundaries.
    - **Load**: Embeds text using embedding-001 and saves vectors to ChromaDB.

---

4. Pipeline Details

4.1. The RAG Pipeline (Information Retrieval)

When a user asks a question about the school (e.g., "What are the majors in the engineering cycle?"):

1. **Decision**: The LLM analyzes the query and determines it needs external knowledge. It generates a tool call for retrieve_esilv_info.

2. **Query Embedding**: The tool converts the user's text query into a vector using Google's embedding-001 model.
3. **Semantic Search**: ChromaDB performs a similarity search (likely Cosine Similarity) to find the top 5 most relevant document chunks (k=5).
4. **Context Construction**: The text content of these 5 chunks is concatenated.
5. **Generation**: The original query + chunks are sent back to the LLM. The LLM synthesizes an answer citing the retrieved facts.

4.2. The Registration Pipeline (Agentic Slot Filling)

When a user expresses interest (e.g., "I want to apply"):

1. **Intent Recognition**: The LLM detects intent to register.
2. **Slot Filling Loop**:
    - The System Prompt instructs the model to gather 3 specific fields: Name, Email, and Interest.
    - The model checks conversation history. If any are missing, it asks the user (e.g., "Could you please provide your email?").
    - This "Ask" action is a standard text response, concluding one turn of the graph.
    - The user responds. The graph runs again.
3. **Tool Execution**: Once all fields are present, the LLM calls save_registration(name, email, interest).
4. **Persistence**: The tool appends the JSON object to data/registrations.json (locking logic is simple file I/O).
5. **Confirmation**: The tool returns "Success", and the LLM informs the user they are registered.

---

5. Evaluation & Results

5.1. Evaluation Methodology

Evaluation of Generative AI systems presents unique challenges. For this project, we utilize a combination of qualitative and functional testing.

- **Retrieval Quality**: Verification that the search returns relevant chunks from the PDF.
    - *Metric*: Relevance of retrieved chunks (Human validation).
- **Response Quality**: ensuring the answer is grounded in the retrieved chunks.
    - *Metric*: Faithfulness (lack of hallucination) and Answer Relevance.
- **Agent Logic**: verifying the state machine correctly transitions through the registration steps.
    - *Metric*: Completion Rate (percentage of "I want to register" intents that result in a saved JSON entry).

5.2. System Results

Scenario A: General Knowledge

- **Input**: "Tell me about the Majors."
- **Behavior**: System retrieves content from Programme_Grande_Ecole.pdf (hypothetical).
- **Result**: The chatbot correctly lists the majors (Financial Engineering, Data & AI, IoT, etc.) as found in the documents.
- **Latency**: Average response time is ~2-4 seconds (dependent on API latency).

Scenario B: Registration

- **Input**: "Register me."
- **Behavior**:
    1. Bot: "Sure! What is your name?"
    2. User: "Solal Chatelain"
    3. Bot: "Thanks Solal. What is your email?"
    4. User: "solal.chatelain@edu.devinci.fr"
    5. Bot: "And your area of interest?"
    6. User: "AI"
    7. Bot calls save_registration.
- **Result**: A new entry adds to registrations.json.

{

"name": "Solal Chatelain",

"email": " solal.chatelain@edu.devinci.fr ",

"interest": "AI"

}

- **Robustness**: The agent successfully handles out-of-order information (e.g., if the user says "I am Solal" initially).

---

6. Challenges and Limitations

6.1. Technical Challenges

- **Context Window**: While Gemini has a large context window, passing too many retrieved chunks (if k is increased) can introduce noise or dilute focus.
- **PDF Parsing**: PyPDFLoader is robust but can struggle with complex layouts (multi-column text, tables inside brochures). This can lead to fragmented text chunks that lose semantic meaning.
- **State Persistence**: The current MemorySaver works in-memory (or simple persistence). For a production app, a persistent database (Postgres/Redis) is needed for long-term history.
- **File Locking**: The JSON registration system is not thread-safe. Concurrent users writing to the file could cause data corruption.

6.2. Hallucination Risks

Even with RAG, there is a risk the model might fill in gaps with general knowledge rather than specific school policy if the retrieval misses the exact document.

---

7. Future Work

7.1. Short-Term Improvements

- **Advanced Ingestion**: Switch to UnstructuredPDFLoader or OCR-based loaders to better handle tables and images in brochures.
- **Database Migration**: Move registrations.json to a SQLite or PostgreSQL database for reliability.

- **Structured Output**: Use structured outputs (JSON mode) for the LLM to ensure the tool arguments are strictly formatted, reducing parsing errors.

7.2. Long-Term Roadmap

- **Multi-Modal RAG**: Since school brochures contain many images and charts, using a simplified multimodal embedding model to retrieve images alongside text would enhance the user experience.
- **Voice Interface**: Integrating Speech-to-Text (STT) and Text-to-Speech (TTS) for a fully vocal kiosk experience on campus.
- **Integration**: Connect the

save_registration tool directly to the school's CRM (Salesforce/HubSpot) instead of a local file.