Project Name: Private & Secure Technical Simplifier For Sensitive Documents

Goal: Creating An Local LLM With Modern LLM Stack (langchain, Rag, Ollama, SLM)

Final-Goal: To Process PDF and Docx inside the local computer without data leaving outside of the computer to ensure the privacy and security

Reason Why I Made This Project?

To address two main pain point:

1. Miss Communication between Technical and non-technical
2. Privacy Concise using LLM
3. Lower Computational Cost

Why I chose Ollama models instead of OpenAI or geminal API key?

The main reason is the project's strict requirement for **data privacy** and **local processing**

Ollama was selected because it **simplifies running models locally**. This ensures that all sensitive technical data processing is done **on-premises**, guaranteeing the data **never leaves the local machine**

Why I Chose SLM (Small Language Models) instead of Stander LLM(Large Language Models)?

The choice of a **Small Language Model (SLM)** over a larger, standard LLM was driven by the project's requirements for **local deployment** and p**erformance**.

- **Efficiency on Local Hardware:** SLMs (like Llama 3.2 Latest) offer the best balance of reasoning capability for complex simplification tasks while remaining **small enough 3B parameter) to run efficiently on local hardware** (typically requiring 2GB+ RAM).
- **Privacy Mandate:** Since the project mandates **local processing** to maintain data privacy, a large LLM would likely be too resource-intensive to run efficiently on a typical local machine, making the smaller, lighter SLM the practical choice

Note: SML Consider an viable solution for long term use

Reference: [Small Language Models are the Future of Agentic AI]

LangChain & Prompt Engineering:

1. LangChain (Orchestration)

Why I Chose It: LangChain was chosen as the **standardized Python framework** to effectively manage the local LLM.

- **Role:** It acts as the "glue" or **orchestration layer** to connect all the components.
- **Proof in Imports:** The imports like **RecursiveCharacterTextSplitter**, **RunnablePassthrough**, **StrOutputParser**, and **ChatOllama** (from your previous context) show how LangChain builds the **simplifier chain**: Prompt -> LLM -> Output.

Prompt Engineering (The Simplification Engine)

Why It's Critical: The **System Instruction** (defined via PromptTemplate in your imports) is the **core logic** that enforces the persona and strict simplification rules.

- **Rules:** It ensures the output is useful by demanding **simple vocabulary, analogies/metaphors**, and critically, the instruction to **"never use jargon or complex sentence structures"**

```
You are a Private & Secure Technical Simplifier. Your role is to translate
        complex technical information from the provided context into
simple,
        non-technical language suitable for a layperson.

        Answer the question based ONLY on the following context.
        If the context does not contain the answer, state that you cannot
find the
        information in the documents. Do not use outside knowledge.

        CONTEXT: {context}
        QUESTION: {question}
```

```
        SIMPLIFIED ANSWER:
        """
```

Why I used Rag (Retrieval-Augmented Generation)?

I used **RAG (Retrieval-Augmented Generation)** as a **conditional feature** to enhance the simplification process when dealing with proprietary data

- **Conditional Need:** RAG is only necessary if the sensitive reports contain **custom, proprietary terms, acronyms, or internal project names** that the base SLM might misunderstand or "hallucinate" about.
- **Purpose:** It **grounds the SLM with private, proprietary facts** (like internal glossaries or documentation) to ensure the simplification of specific jargon is accurate.
- **Privacy-First Tool Choice:** The RAG system uses **LangChain + ChromaDB**. ChromaDB was chosen because it runs locally and is file-based, making it a **local and secure** solution that maintains the project's strict privacy mandate. You specifically **avoided cloud-based** vector databases (like Pinecone) to prevent storing embeddings on a third-party cloud service.

Technical Problem Solving & Project Refinement

This section summarizes the critical debugging, cleanup, and code modernization tasks performed, demonstrating strong command-line, Git, and LLM framework expertise.

1. Git & Environment Cleanup (Addressing "The 10K Problem")

A significant portion of the initial effort involved fixing repository issues caused by incorrect Git initialization and configuration.

Issue	Cause/Symptom	Resolution/Fix
Git Initialization Scope Error (The 10K Problem)	Accidentally running git init in a parent directory (e.g., User or Desktop) , causing Git to track approximately **10,000 files**	Located and forcefully deleted the incorrect, hidden .git folder using powerful terminal commands (DEL /F /S /Q /A .git*.* and RMDIR /S /Q .git).

File Deletion Attribute Error	Windows reported it "cannot find the file specified" because Git set the .git folder as a hidden/read-only system folder.	Used the correct, forceful command syntax (DEL /F /S /Q /A and RMDIR /S /Q) to bypass file attributes.
Virtual Environment Tracking Error	Attempted a massive commit **before** the .gitignore file was active, causing Git to track thousands of files inside .venv_312 and env folders	Corrected .gitignore, then used git reset --soft HEAD^ (undo massive commit) and git rm -r --cached . (clean staging) to commit only source code
Line Ending Warnings (CRLF vs. LF)	Git detected mixed line endings common inside virtual environment files	Eliminated the source of numerous warnings by removing the virtual environment from tracking
Dependency Hell	The risk of conflicting package versions.	**AVOIDED** by consistently using a dedicated **virtual environment** (.venv_312) to isolate project dependencies

2. LLM Framework Modernization (rag_cli.py changes)

The main application code was updated to adopt modern LangChain standards, enhance performance, and ensure privacy.

- **Imports Consolidated:** Updated and consolidated imports to use the modern, split packages, specifically using langchain_ollama for both OllamaEmbeddings and OllamaLLM.
- **Real-time Streaming:** Implemented StdOutCallbackHandler(BaseCallbackHandler) and instantiated the LLM with callbacks to provide **real-time streaming** output in the CLI.
- **Privacy Enhancement:** Explicitly set a default environment variable to **disable Chroma telemetry** (os.environ.setdefault("CHROMA_TELEMETRY_ENABLED", "false")).
- **Code Cleanup:** Removed the use of deprecated Ollama classes and removed vector_store.persist() as Chroma now auto-persists.