Approaching the Problem Statement

When working on **automating the workflow of an orchestrator and synthesizer using LangGraph**, the first challenge was to identify a **real-world use case** where this architecture would be useful. Since the goal is to process structured tasks using **open-source LLM models**, we needed a domain that requires heavy text analysis, transformation, and structured outputs.

After exploring different possibilities, I decided to focus on a **Legal Text Regenerator**.

Why Choose Legal Text Regeneration?

Legal documents are often complex, lengthy, and difficult to understand. Lawyers, businesses, and individuals frequently need to:

- Rewrite contracts to match different tones or simplify the language.
- Extract key clauses from a legal document.
- Identify risks and potential compliance issues.
- Generate alternative legal arguments and contract suggestions.
- Format and structure legal documents for easy readability.

Since this involves **multiple interconnected tasks**, LangGraph provides a great way to model this workflow, and **LangSmith helps debug each step efficiently**.

Step-by-Step Breakdown of the Solution

1. Understanding the Orchestrator and Synthesizer

Before jumping into the implementation, I first needed to clearly define:

- **Orchestrator**: The system that manages the entire workflow, ensuring that each step executes in the correct sequence.
- **Synthesizer**: The part that actually generates the outputs using an **open-source LLM** (like Llama3 from Groq).

In simple terms:

- The **Orchestrator** decides when to rewrite, summarize, extract, or analyze the text.
- The **Synthesizer** (LLM) takes inputs from each step and generates meaningful responses.

2. Setting Up the Environment

Since we are using Groq's Llama3-8B model, we need to load API keys and initialize the model.

- Used **dotenv** to load API keys securely.
- Set up **LangSmith** for debugging each step.
- Initialized the **LLM** to be used in the synthesizer.

3. Defining the Workflow Components

To process a legal document, I divided the workflow into distinct steps:

1. **Rewrite Legal Text**: Adjusts the tone and complexity of the document.
2. **Summarize the Document**: Extracts the main points in a concise form.
3. **Extract Key Clauses**: Identifies important legal clauses automatically.
4. **Risk Analysis**: Detects potential legal risks in the document.
5. **Check Compliance**: Ensures that the document follows legal guidelines.
6. **Suggest Contract Improvements**: Provides ways to enhance the document.
7. **Generate Legal Arguments**: Proposes alternative legal justifications.
8. **Format the Document**: Organizes the document for clarity and readability.
9. **Compare Original vs. Rewritten Text**: Highlights key differences between the original and rewritten versions.

4. Implementing with LangGraph

LangGraph is perfect for **structuring workflows** where different steps depend on previous ones.

- Defined a **StateGraph** to model the workflow.
- Added nodes for each of the functions (rewrite, summarize, analyze risks, etc.).
- Connected them using **edges**, ensuring that each function gets the required inputs before execution.

Example:

- The **rewriting step** happens first.
- Once rewritten, the same text is used for summarization, clause extraction, and risk analysis **in parallel**.
- Compliance checking, contract improvements, and formatting happen later in the sequence.

5. Debugging with LangSmith

Since multiple steps depend on previous results, debugging is crucial.

- Used **LangSmith tracing** to check how the text flows through the graph.
- If a function fails or returns an unexpected output, LangSmith makes it easy to debug where things went wrong.

6. Creating a User-Friendly Interface with Streamlit

To make this system accessible, I built a simple **Streamlit app** where:

- Users can enter a **legal text**.
- Select the desired **tone** (formal, empathetic, neutral, strength-based).
- Click a **button** to process the text.
- See results for **each step** (rewritten text, key clauses, risk analysis, compliance check, etc.).

This provides an easy way to test and use the application without running code manually.

Final Thoughts

This project **automates legal text processing** using LangGraph and **open-source LLMs** while ensuring clarity, risk detection, and compliance.

- **LangGraph** manages the workflow efficiently.
- **Groq's Llama3 model** provides intelligent text generation.
- **LangSmith** helps debug each step effectively.
- **Streamlit** makes the app user-friendly.

This approach can be **extended** to other domains like **medical documents, business reports, or educational content** where structured text processing is needed.