Resume Tailor Agent: System Design and LLM Integration Strategy

1. Introduction

This document outlines the system design and LLM integration strategy for the Resume Tailor Agent. The primary goal of this project is to provide a robust and flexible tool for tailoring resumes to specific job descriptions while preserving original formatting. The system is designed to support both local (Ollama) and API-based (OpenAI, Anthropic) Large Language Models (LLMs), allowing for easy switching between different LLM providers based on user preference or future requirements.

2. System Architecture

The Resume Tailor Agent follows a modular and layered architecture, promoting separation of concerns, maintainability, and extensibility. The core components are designed to interact seamlessly, ensuring efficient processing of resumes and job descriptions, and effective utilization of LLMs for content generation. The architecture can be broadly divided into the following key modules:

2.1. User Interface (CLI/Future Web UI)

Currently, the primary interface for the Resume Tailor Agent is a Command Line Interface (CLI), as indicated by the `main.py` script and its argument parsing. This allows users to specify input files (resume, job description) and output locations, as well as select the LLM model to use. In the future, as per the roadmap, a Streamlit web interface will be developed to provide a more user-friendly graphical interface, enabling easier interaction for a broader audience.

2.2. Data Management Layer

This layer is responsible for handling input and output files, ensuring that the system can read resumes and job descriptions in specified formats and save the tailored resumes. It includes:

- **Input Handling:** The system reads resume files in `.docx` format and job descriptions from `.txt` files or directly from input strings. This requires robust parsing capabilities to extract relevant text content while preserving structural information.
- **Output Generation:** After processing, the system generates a new `.docx` resume with the tailored content, crucially maintaining the original formatting of the input resume.

This implies a sophisticated document manipulation capability that can selectively update text sections without altering the document's layout or styles.

2.3. Core Processing Logic (`main.py`)

The `main.py` script serves as the orchestrator of the entire application. Its responsibilities include:

- **Argument Parsing:** Interpreting command-line arguments to configure the application's behavior (e.g., input/output paths, LLM model selection).
- **File Loading:** Loading the resume and job description files into memory for processing.
- **Prompt Construction:** Dynamically building prompts for the LLM based on the extracted resume sections and the job description. This is a critical step that influences the quality and relevance of the LLM's output.
- **LLM Invocation:** Calling the `llm_interface.py` module to interact with the selected LLM.
- **Content Integration:** Receiving the tailored content from the LLM and integrating it back into the resume document.
- **File Saving:** Saving the final tailored resume to the specified output location.

2.4. Resume Parsing and Updating Module (`resume_parser.py`)

This module is central to the system's ability to interact with `.docx` files. It encapsulates the logic for extracting specific sections from a resume and for updating those sections with new content while preserving formatting. Key functionalities include:

- **Section Extraction:** Identifying and extracting key sections from the `.docx` resume, such as the summary, skills, and experience sections. This likely involves parsing the document structure and identifying patterns or markers for these sections.
- **Content Update:** Replacing the extracted sections with the LLM-generated content. The challenge here is to perform in-place updates without disrupting the document's original formatting, which might require advanced document processing libraries.

2.5. LLM Interface Module (`llm_interface.py`)

This module acts as an abstraction layer for interacting with various LLMs, whether local or remote. This is a crucial component for achieving the flexibility to switch between LLMs with minimal code changes. Its responsibilities include:

- **Model Abstraction:** Providing a unified interface (`run_llm(prompt, model=

```
Python
```

```
run_llm(prompt, model="local")
```

`)` for different LLM providers. This module will handle the specifics of API calls for remote LLMs (OpenAI, Anthropic) and interaction with the local Ollama instance.

- **API Key Management:** Securely managing API keys for paid LLM services, likely through environment variables as suggested in the README.
- **Error Handling:** Implementing robust error handling for LLM interactions, including network issues, API rate limits, and invalid responses.

2.6. Utility Functions (`utils.py`)

This module will house various helper functions that support the operations of other modules. This promotes code reusability and keeps the main logic clean. Examples of utility functions might include text cleaning, string manipulation, or file path handling.

3. LLM Integration Strategy

The core principle of the LLM integration strategy is **abstraction**. By centralizing all LLM interactions within `llm_interface.py`, the system achieves high flexibility and ease of switching between different LLM providers.

3.1. Unified Interface

The `run_llm` function will serve as the single point of contact for all LLM requests. It will take the prompt and the desired model as arguments. Based on the `model` argument, it will internally route the request to the appropriate LLM client (Ollama, OpenAI, Anthropic).

3.2. Dynamic Model Selection

The `main.py` script will pass the chosen LLM model (e.g., 'local', 'openai', 'anthropic') to `llm_interface.py`. This dynamic selection allows users to easily configure the LLM at runtime via command-line arguments, fulfilling the requirement of easy switching between LLMs.

3.3. API Key Management

For paid LLM services (OpenAI, Anthropic), API keys will be read from environment variables (`OPENAI_API_KEY`, `ANTHROPIC_API_KEY`). This is a standard and secure practice for managing sensitive credentials, preventing them from being hardcoded into the application.

3.4. Local LLM Integration (Ollama)

Integration with Ollama will involve making HTTP requests to the local Ollama server. The `llm_interface.py` will construct the appropriate request payload (including the model name and prompt) and send it to the Ollama API endpoint. This allows for completely free and local processing, as highlighted in the README.

3.5. Remote LLM Integration (OpenAI, Anthropic)

For OpenAI and Anthropic, the `llm_interface.py` will utilize their respective Python client libraries (if available and suitable) or directly make HTTP requests to their APIs. This ensures compatibility with their services and allows access to their advanced models like GPT-4 and Claude 3.5 Sonnet.

3.6. Prompt Engineering

The quality of the tailored resume heavily depends on the effectiveness of the prompts sent to the LLM. The `main.py` will be responsible for constructing clear, concise, and effective prompts that guide the LLM to rewrite the summary, skills, and experience sections accurately and appropriately for the given job description. This will involve:

- **Contextual Information:** Including the original resume content and the job description in the prompt.
- **Instruction Clarity:** Providing explicit instructions on what sections to rewrite and what style to adopt.
- **Formatting Preservation:** Emphasizing the need to maintain the original formatting of the resume sections.

4. Data Flow

The following steps illustrate the typical data flow within the Resume Tailor Agent:

1. **User Input:** The user provides the path to their `.docx` resume, the `.txt` job description, and the desired output path and LLM model via command-line arguments.
2. **File Loading:** `main.py` loads the `.docx` resume and `.txt` job description.
3. **Resume Parsing:** `resume_parser.py` extracts the summary, skills, and experience sections from the loaded `.docx` resume.
4. **Prompt Generation:** `main.py` constructs a comprehensive prompt using the extracted resume sections and the job description.
5. **LLM Invocation:** `main.py` calls `llm_interface.py` with the generated prompt and the selected LLM model.

6. **LLM Processing:** `llm_interface.py` sends the prompt to the chosen LLM (local Ollama or remote API). The LLM processes the prompt and returns the rewritten content for the specified sections.
7. **Content Integration:** `main.py` receives the rewritten content from `llm_interface.py`.
8. **Resume Update:** `resume_parser.py` updates the original `.docx` resume with the new content, ensuring that the original formatting is preserved.
9. **Output Saving:** `main.py` saves the modified `.docx` resume to the specified output path.

5. Future Enhancements and Scalability

The current design lays a solid foundation for future enhancements and scalability, as outlined in the roadmap.

5.1. Extending Tailoring to Experience Bullets

The current design already accounts for extending tailoring to experience bullets. The `resume_parser.py` will need to be enhanced to accurately identify and extract individual experience bullets, and the prompt engineering in `main.py` will need to be refined to instruct the LLM on how to rewrite these bullets effectively while maintaining their structure and context.

5.2. Cover Letter Generator

Adding a cover letter generator would involve:

- **New Prompting Strategy:** Developing a new set of prompts specifically designed for generating cover letters, taking into account the job description and key aspects of the resume.
- **Output Format:** Generating the cover letter in a suitable format (e.g., `.docx` or `.txt`).
- **Integration with LLM:** Utilizing the existing `llm_interface.py` to interact with LLMs for cover letter generation.

5.3. Streamlit Web Interface

Building a Streamlit web interface will significantly improve user experience. This would involve:

- **Frontend Development:** Creating a web-based UI using Streamlit, allowing users to upload files, input text, select LLM models, and view tailored resumes.
- **Backend Integration:** Connecting the Streamlit frontend to the existing Python backend logic (`main.py`, `resume_parser.py`, `llm_interface.py`).

- **Deployment:** Considering deployment options for the Streamlit application to make it accessible online.

5.4. Job Application Tracker

Implementing a job application tracker would involve:

- **Database Integration:** Introducing a database (e.g., SQLite for simplicity, or a more robust solution for scalability) to store application details (company, role, date applied, status, tailored resume link).
- **New UI Components:** Adding UI elements in the Streamlit interface to manage and view application records.
- **Reporting:** Generating reports or visualizations based on application data.

6. Conclusion

The proposed system design for the Resume Tailor Agent provides a modular, flexible, and extensible framework for automated resume tailoring. The emphasis on a clear LLM abstraction layer ensures that the system can easily adapt to new LLM technologies and user preferences. With a well-defined architecture and a clear roadmap for future enhancements, this project is poised to become a valuable tool for job seekers. The design also considers the initial free usage with Ollama and the seamless transition to paid API services, offering a cost-effective and powerful solution. This comprehensive approach ensures that the project is not only functional but also scalable and maintainable in the long term.