AI Powered Document Editing Assistant

Introduction:

In today's fast-paced digital landscape, efficiency, and user experience are paramount, especially in workflows that involve frequent and repetitive document editing. While many of us have experienced the productivity boost of vibe-coding, where AI understands and executes our coding intentions, such simplicity is still far less prevalent in the domain of document editing. This project bridges that gap.

We have developed an **AI-powered document editing assistant**, designed to enable users to interact with documents using natural language commands. Instead of navigating complex interfaces, toggling multiple toolbars, or repeatedly applying formatting and layout changes, users can simply describe what they want to modify or create. The system translates their intent into real-time edits, allowing them to see the documents evolve interactively.

Whether it's inserting a new table, updating a pricing section, or adding contractual clauses, the assistant is capable of understanding nuanced instructions and applying them precisely to the document. Users no longer need to spend hours manually adjusting formatting or hunting for the right option, they can focus on *what* they want, and let the AI handle the *how*.

There is a powerful agent-tool architecture at the core of the system. The agent is responsible for interpreting user intent through natural language understanding, while the tools act as modular executors, each specialized in handling a specific type of document manipulation (e.g., editing headers, updating body sections, or fetching reusable templates). This modular design not only ensures scalability but also makes the system robust and adaptable to a variety of document formats and structures.

This solution is aimed at businesses that rely heavily on document creation and editing workflows, such as legal, sales, procurement, and operations teams. These organizations often work with structured documents like contracts, agreements, proposals, and price guides, where even minor changes can be tedious and time-consuming. Our system dramatically reduces turnaround time, improves accuracy, and allows for rapid iteration, all with minimal training or onboarding.

Client details:

Name: Confidential | **Industry**: Construction, Software | **Location**: USA

AI Powered Document Editing Assistant

Technologies:

Agents, Tool Calling, OpenAI Assistants, GPT-4o-mini, LangChain, Pydantic, Redis, Parse DB, FastAPI, EC2, ECS.

Project Description:

The objective was to develop a document editor **agent** capable of processing natural language instructions and executing precise modifications on a complex, deeply nested JSON-based document structure. The backend is designed using FastAPI, and the frontend is a React-based UI that renders the updated document in real time.

1. **Agent-Tool Orchestration:**

   At the core of the system lies **LangChain custom agent** powered by an LLM, and configured with a suite of domain-specific instructions and tools designed to operate on a deeply nested JSON structure. These tools support both high level structural modifications such as updating the page configuration, renaming a document or programmatically inserting multiple header/body/signature sections and low-level edits including updating specific cell value (that will be mapped to a tabular structure), modifying date display formats, toggling boolean configuration flags and applying granular styling like font size, color and text emphasis (e.g., bold, underline or italics).

   Each operation is encapsulated as a **LangChain BaseTool**, with a well-defined input schema modeled using Pydantic, enabling strict validation, type safety, and seamless integration with the agent runtime. These tools are enriched with custom logic tailored for domain-specific tasks, such as fetching the list of available contract templates, selecting a target document (represented as a nested JSON object), and using that selection as context for subsequent editing operations. Once a template is selected, it serves as the central data structure that flows through the pipeline. Tools such as *AddHeaderSectionTool* and *UpdateHeaderSectionTool* implement structured CRUD operations, enabling dynamic modification of the document's layout and content hierarchy in a controlled, composable manner.

   All the tools are dynamically registered to the LangChain agent and are designed for context-aware execution. During tool invocation, critical contexts such as the selected document (complex JSON object), the user's session ID, and the unique template ID are passed explicitly, and relevant metadata such as document name and font name are extracted by the LLM and passed implicitly to ensure precise operation. This setup is further enhanced with session-aware memory using RunnableWithMessageHistory

implementation. Together, they enable coherent multi-turn conversations by preserving chat history per session, allowing agents to maintain state, track document-specific operations, and respond in a contextually intelligent manner.

This pipeline also supports **tool chaining,** where the output of one tool (e.g., selected template) feeds into the next operation (e.g., modifying the attributes of the body section), enabling multi-turn interactions across a session. To support real-time responsiveness and non-blocking operations, tools interacting with external systems (such as the DB where the documents are stored as JSON objects) are implemented using async/await patterns; ensuring input/output tasks do not block agent execution.

2. **Handing Document data:**

   Document data represented as a complex JSON object is stored in a database; hence handling this structure involves three key operations:

   a. **Fetching from Database:** Templates are stored in a NoSQL-like document database (e.g., MongoDB/Parse DB). Upon receiving a valid template ID, the server fetches the corresponding JSON structure. This is facilitated through a FastAPI API endpoint that handles authentication and payload validation.
   b. **Performing in-memory modifications:** The tools mutate specific sections of the documents (JSON) by adding rows, modifying text fields, etc. All these modifications are done maintaining the original schema of the document.
   c. **Writing back to the database:** After editing, the modified JSON is validated and persisted back to the DB. The backend also maintains an edit history by storing the last modified user name, timestamps, and names of the changed fields.

3. **Conversational Agent API:**

   The backend provides a set of streaming FastAPI endpoints, each secured via session-based authentication using a session ID. These endpoints accept natural language user queries, invoke the designed agent to process the request, and stream the generated responses back to the client. Additionally, the API is designed to detect and return the template ID when applicable, enabling downstream interactions to reference and operate on the correct document context.

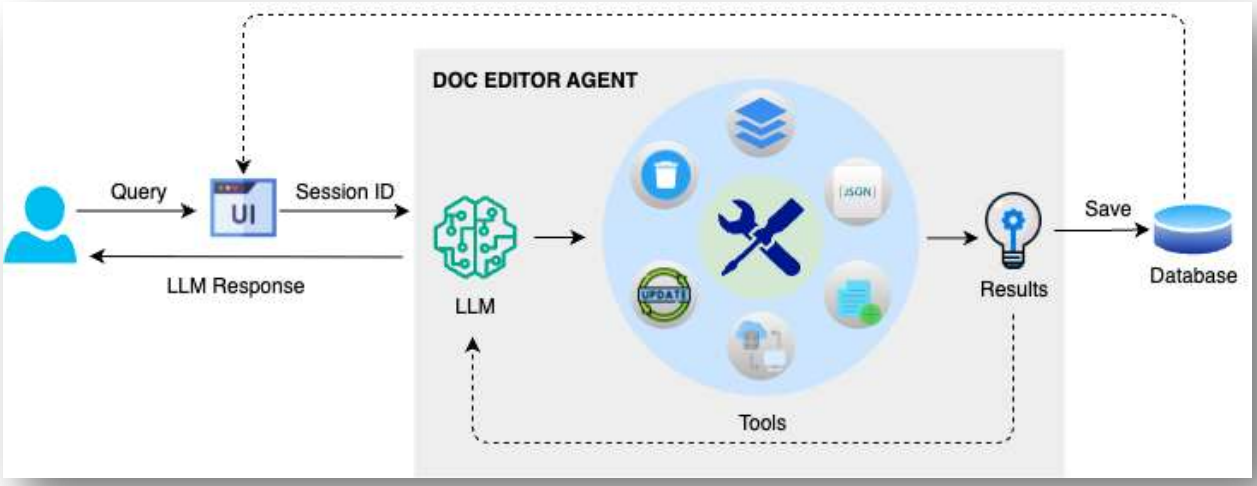
4. **React-Based UI:**

   The front end consumes the updated, deeply nested JSON structure and renders it into a dynamic, schema-driven, and editable document interface built using React. Key features of this UI layer include:

AI Powered Document Editing Assistant

- **Section-wise component rendering**: The JSON is parsed into logical sections, each mapped to modular React components, enabling granular control and dynamic layout management.
- **State sync**: A robust local state management system ensures that the front end stays in sync with backend updates.
- **Dual Interaction Models:** Users can either edit fields directly via interactive form elements or issue natural language queries through an integrated chatbot interface.
- **Live-preview:** Leveraging streaming responses and reactive state binding, the UI provides a live preview of the document, instantly reflecting JSON updates as they are processed by the backend.

Pipeline:



Variables impacting the performance of the application:

1. Tool execution time and custom logic inside tools can affect the time taken by each tool and hence, the response generation.
2. The response time of the LLM model can cause latency issues.
3. Tool invocation complexity, number of tool calls and nested invocations, and chains can increase cumulative latency.
4. The size of the JSON document, with increasingly deeply nested JSONs, increases the time required for CRUD operations.