DocToJSON - An Architecture for Data Extraction from Unstructured Text

A significant portion of our B2B data ingestion workflows involves the manual or semi-automated transformation of unstructured data sources (e.g., PDFs, DOCX, email threads) into highly structured, machine-readable JSON formats. These workflows are frequently bottlenecks due to two primary technical constraints:

1. **Schema Complexity:** Target schemas often exhibit high dimensionality, with deep nesting (3-7 levels). This complexity makes it difficult to construct a single, coherent prompt for a Large Language Model (LLM) that doesn't degrade generation quality.
2. **Context Window Limitations:** Input documents frequently exceed the context window of modern LLMs, making a single-pass approach infeasible.

This proposal outlines a system architecture designed to overcome these limitations through a composable, multi-stage pipeline.

Proposed Architecture: A Decomposed RAG-based System

The proposed solution abandons a monolithic, single-prompt strategy in favor of a more robust, **divide and conquer** architecture built on two core principles: **Schema Decomposition** and **Retrieval-Augmented Generation (RAG)**.

Part 1: Schema Decomposition

Mechanism: The system will traverse the master schema's Abstract Syntax Tree (AST), isolating top-level objects and their children into independent, self-contained schema definitions.

Rationale: This transforms a single, high-complexity generation task into multiple low-complexity tasks. Each sub-task requires the LLM to focus on a smaller, more manageable output structure, which has been shown to significantly improve generation accuracy and adherence to format. This also allows for parallelization of the extraction process.

Part 2: Retrieval-Augmented Generation (RAG)

Mechanism: The source document will be chunked, embedded using a text embedding model and indexed in a vector database.
Rationale: Instead of passing the entire document as context, the system will perform a semantic search against the vector store to retrieve only the most relevant text chunks for each sub-schema. This decouples the system's performance from the input document's size and ensures the LLM's context window is utilized efficiently.

System Workflow & Implementation Details

The end-to-end process is executed via a four-stage pipeline:

1. **Phase 1: Schema Decomposition & Prep:** This schema is used only for defining the output structure. It is parsed and decomposed into sub-schemas, and for each, a corresponding Pydantic model is dynamically generated in memory to serve as a strict validation and data coercion layer.
2. **Phase 1: Ingestion & Decomposition:** The unstructured source document is ingested into the vector database. This document is used *only* for information retrieval. It is loaded, chunked, vectorized, and indexed into the vector store.

3. **Phase 3: Orchestrated Extraction & Validation:** An orchestrator module iterates through each sub-schema defined in Phase 1:

   a. **Context Retrieval:** It generates a semantic query from the sub-schema's key and field descriptions to retrieve the top-k relevant chunks *from the indexed source document*.

   b. **Prompt Engineering:** A targeted prompt is constructed, containing the instruction, the sub-schema definition, and the retrieved text chunks as context.

   c. **LLM Call:** The prompt is sent to a generation model with a JSON mode enabled.

   d. **Validation:** The LLM's output is immediately parsed and validated against the dynamically generated Pydantic model. On validation failure, a retry policy can be triggered, potentially feeding the validation error back to the LLM for self-correction.

4. **Phase 4: Composition:** The validated Pydantic objects are programmatically re-assembled into the final, composite JSON structure, adhering to the master schema.

Technical Risks & Mitigation Strategies

Bottleneck 1: System performance is critically dependent on the RAG pipeline's ability to retrieve the correct context.

Bottleneck 2: The initial design assumes sub-schemas are contextually independent. This may not hold for documents where information is highly relational.

How We Can Make It Even Better Later

This architecture is built to grow. Down the road, we can easily add features like:

- **Confidence Scores:** We can ask the AI to rate how confident it is about each piece of data it extracts. This would let us automatically flag things for a quick human review.
- **A Final Review:** We could add a final llm-as-a-judge step, where another AI call checks the finished JSON against the original document for any inconsistencies.

This approach gives us a powerful and scalable way to solve our data extraction headaches. It's a solid foundation that tackles the biggest challenges head-on and opens the door for powerful new features in the future. I'm looking forward to discussing it with you all.