Multi-modal Partitioning for RAG

RAG on documents with text, tables, images and code blocks.

7 min read · Just now

Yogesh Haribhau Kulkarni (PhD)

Listen Share ••• More

Photo by Markus Spiske on Unsplash

Traditional Retrieval Augmented Generation (RAG) systems operate under the assumption that documents are homogeneous text entities, leading to significant information loss when processing real-world documents containing heterogeneous content modalities. This technical deep-dive explores the implementation of a multi-modal RAG system that addresses the fundamental limitations of conventional approaches by implementing modality-specific parsing, semantic description generation, and specialized retrieval mechanisms.

The Problem: Limitations of Traditional RAG Architecture

Information Loss in Heterogeneous Documents: Traditional RAG implementations suffer from a critical architectural flaw: they treat all document content as uniform text, regardless of its semantic structure or modality. When processing documents containing tables, images, code blocks, or structured data, conventional systems either:

1. **Ignore non-text content entirely**, resulting in significant information loss
2. **Convert all content to plain text**, destroying semantic relationships and structural information
3. **Apply uniform chunking strategies**, breaking logical content boundaries across modalities

Consider a technical document containing a performance comparison table followed by architectural diagrams and implementation code. A traditional RAG system would either skip the visual elements entirely or convert the table to poorly formatted text, losing the structured relationships that make tabular data queryable and meaningful.

Semantic Fragmentation in Uniform Chunking: Standard chunking approaches use fixed-size or sentence-based segmentation, which creates several problems:

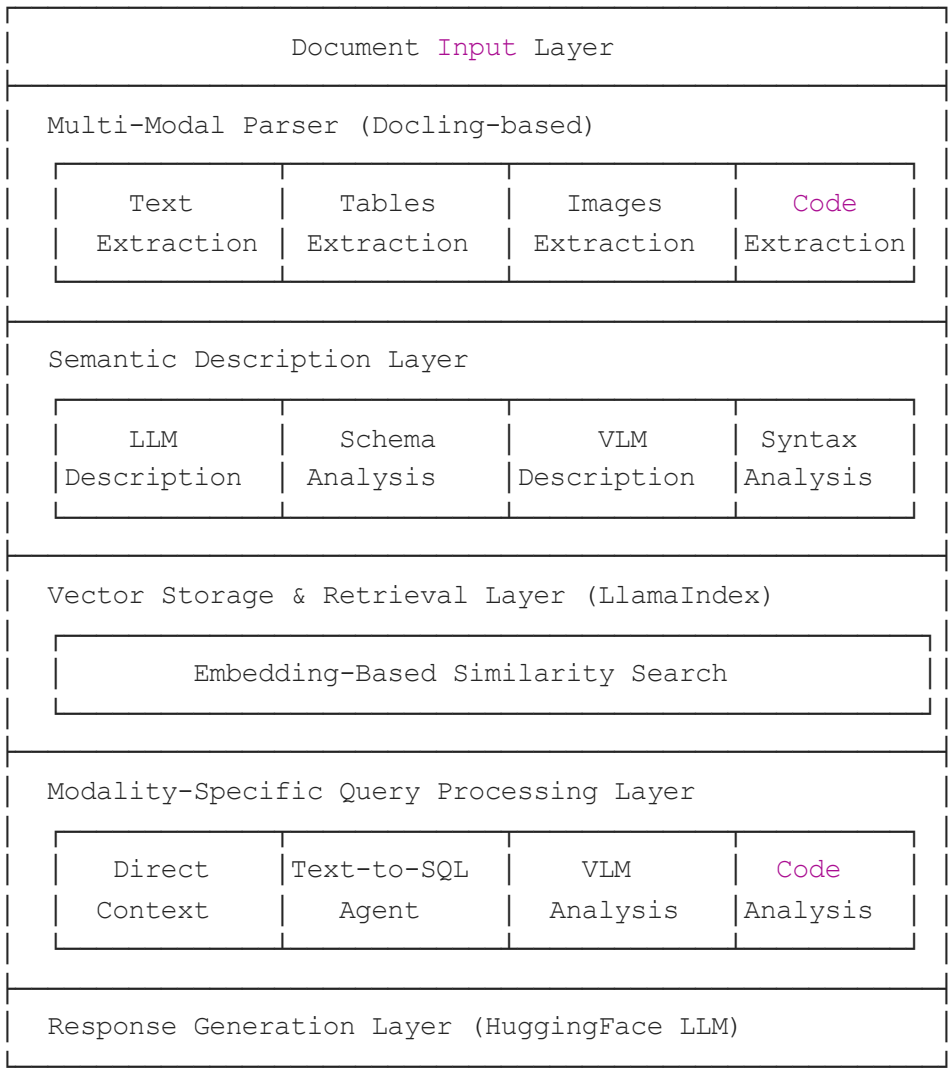
```
# Traditional approach - semantic boundaries are ignored
def naive_chunking(text, chunk_size=512):
    chunks = []
    for i in range(0, len(text), chunk_size):
        chunks.append(text[i:i+chunk_size])
    return chunks
```

This approach fragments logical units like tables, code blocks, and image captions, making it impossible to maintain contextual coherence during retrieval.

Query-Response Mismatch for Structured Data: When users query structured content (“What was the average revenue in Q3?” or “Show me the database schema”), traditional RAG systems cannot leverage the structured nature of the data, resulting in imprecise or incorrect responses.

The Solution: Multi-Modal RAG with Semantic Partitioning

High-Level Architecture Overview: Our multi-modal RAG system implements a three-tier architecture addressing these fundamental limitations:



Modality-Specific Chunk Architecture

The system implements a hierarchical chunk taxonomy using Pydantic models to maintain type safety and semantic integrity:

```
class ChunkType(Enum):
    TEXT = "text"
    TABLE = "table"
    IMAGE = "image"
    CODE = "code"

class BaseChunk(BaseModel):
    chunk_id: str = Field(description="Unique identifier for the chunk")
```

```
    chunk_type: ChunkType = Field(description="Type of content chunk")
    description: str = Field(default="", description="AI-generated descriptic
    source_page: Optional[int] = Field(default=None, description="Source page
    bbox: Optional[Dict[str, float]] = Field(default=None, description="Bound
```

Each specialized chunk type extends the base with modality-specific attributes:

```
class TableChunk(BaseChunk):
    table_data: List[List[str]] = Field(description="Table data as list of rc
    headers: List[str] = Field(description="Column headers")
    table_html: Optional[str] = Field(default=None, description="HTML represe
    chunk_type: ChunkType = Field(default=ChunkType.TABLE)
    num_rows: Optional[int] = Field(default=None, description="Number of rows
    num_cols: Optional[int] = Field(default=None, description="Number of colu
```

Implementation Deep Dive

Multi-Modal Document Parsing with Docling Integration: The parsing layer leverages Docling's advanced document understanding capabilities to maintain semantic boundaries:

```
class DoclingParser:
    def parse_document(self, document_path: str) -> List[Union[TextChunk, Tab
        logger.info(f"Starting document parsing: {document_path}")

        # Convert document using docling
        result: ConversionResult = self.converter.convert(document_path)
        document = result.document

        chunks = []

        # Extract each modality while preserving semantic boundaries
        text_chunks = self._extract_text_chunks(document)
        table_chunks = self._extract_table_chunks(document)
        image_chunks = self._extract_image_chunks(document)
        code_chunks = self._extract_code_chunks(document)

        chunks.extend([text_chunks, table_chunks, image_chunks, code_chunks])

        # Generate AI-powered semantic descriptions
        chunks_with_descriptions = self._generate_descriptions(chunks)
```

```
        return chunks_with_descriptions
```

AI-Powered Semantic Description Generation: The system implements modality-specific description generation using specialized models:

```
def _generate_descriptions(self, chunks: List[BaseChunk]) -> List[BaseChunk]:
    for chunk in chunks:
        if isinstance(chunk, TextChunk):
            chunk.description = self._generate_text_description(chunk.text_cc
        elif isinstance(chunk, TableChunk):
            chunk.description = self._generate_table_description(chunk)
        elif isinstance(chunk, ImageChunk):
            chunk.description = self._generate_image_description(chunk)
        elif isinstance(chunk, CodeChunk):
            chunk.description = self._generate_code_description(chunk.code_cc

    return chunks
```

For table chunks, the description includes structural analysis:

```
def _generate_table_description(self, table_chunk: TableChunk) -> str:
    headers_str = ", ".join(table_chunk.headers) if table_chunk.headers else
    description = f"Table with {table_chunk.num_rows} rows and {table_chunk.n
    description += f"Column headers: {headers_str}. "

    if table_chunk.table_data and len(table_chunk.table_data) > 0:
        sample_row = table_chunk.table_data[0][:3]
        description += f"Sample data: {', '.join(map(str, sample_row))}..."

    return description
```

Vector-Based Retrieval with Metadata Preservation: The retrieval layer embeds only the semantic descriptions while preserving full content as metadata:

```
def ingest_chunks(self, chunks: List[BaseChunk]):
    documents = []
    for chunk in chunks:
        # Embed descriptions for semantic search
        document = Document(
            text=chunk.description,
            metadata={
                'chunk_id': chunk.chunk_id,
                'chunk_type': chunk.chunk_type.value,
                'source_page': chunk.source_page,
                'bbox': chunk.bbox,
                # Store full content for modality-specific processing
                'content': self._serialize_chunk_content(chunk)
            }
        )
        documents.append(document)

    # Create vector index with semantic descriptions
    self.vector_index = VectorStoreIndex.from_documents(documents, storage_cc
```

Modality-Specific Query Processing: The system implements specialized processing for each retrieved chunk type:

```
def _process_retrieved_node(self, node_with_score: NodeWithScore, query: str)
    chunk_type = node_with_score.node.metadata.get('chunk_type')
    content = node_with_score.node.metadata.get('content', {})

    if chunk_type == ChunkType.TEXT.value:
        return f"Text Content:\n{content.get('text_content', '')}"

    elif chunk_type == ChunkType.TABLE.value:
        # Use specialized table query agent for structured queries
        table_chunk = self._reconstruct_table_chunk(metadata, content)
        table_result = self.table_agent.query_table(table_chunk, query)
        return f"Table Data:\n{table_result}"

    elif chunk_type == ChunkType.IMAGE.value:
        # Return enhanced image description and metadata
        return f"Image Content:\n{self._format_image_context(content)}"

    elif chunk_type == ChunkType.CODE.value:
        # Return formatted code with syntax highlighting
        return f"Code Content:\n```{content.get('programming_language', '')}\
```

Text-to-SQL Agent for Table Queries: For table chunks, the system implements a sophisticated text-to-SQL conversion mechanism:

```
class TableQueryAgent:
    def query_table(self, table_chunk: TableChunk, query: str) -> str:
        # Create temporary SQLite database
        self._create_temp_table(table_chunk)

        # Generate SQL query from natural language
        sql_query = self._generate_sql_query(table_chunk, query)

        # Execute SQL query and return formatted results
        result = self._execute_sql_query(sql_query)

        return result

    def _generate_sql_query(self, table_chunk: TableChunk, query: str) -> str
        schema_info = f"Table schema: {', '.join(table_chunk.headers)}"
        prompt = f"""Convert this question to SQL query:
Table: query_table
{schema_info}
Question: {query}
SQL:"""

        # Use LLM to generate SQL from natural language
        inputs = self.tokenizer.encode(prompt, return_tensors="pt")
        outputs = self.model.generate(inputs, max_length=inputs.shape[1] + 50
        sql_query = self.tokenizer.decode(outputs[0], skip_special_tokens=Tru

        return sql_query
```

Running the System

Installation and Setup

```
# Clone and setup environment
git clone https://github.com/yogeshhk/TeachingDataScience
cd Code/multimodal_chatbot
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Basic Usage

```
from llamaindex_rag import RAGPipeline

# Initialize pipeline with custom models
pipeline = RAGPipeline(
    embedding_model="sentence-transformers/all-mpnet-base-v2",
    llm_model="microsoft/DialoGPT-medium"
)

# Process document end-to-end
stats = pipeline.process_document("technical_report.pdf")
print(f"Processed: {stats}")

# Query different content types
text_response = pipeline.query_document("What are the main conclusions?")
table_response = pipeline.query_document("What was the average performance in
image_response = pipeline.query_document("Describe the architecture diagram")
code_response = pipeline.query_document("Show me the implementation of the sc
```

Advanced Configuration

```
from docling_parsing import DoclingParser
from llamaindex_rag import MultiModalRAG

# Custom parser configuration
parser = DoclingParser(
    text_model_name="microsoft/DialoGPT-medium",
    vision_model_name="microsoft/git-large",
    device="cuda"
)

# Custom RAG configuration
rag = MultiModalRAG(
    embedding_model_name="sentence-transformers/all-mpnet-base-v2",
    llm_model_name="microsoft/DialoGPT-medium",
    device="cuda"
)
```

Advantages

1. Semantic Preservation

- Maintains logical boundaries between different content types
- Preserves structural relationships in tables and code

- Retains visual context for images and diagrams

2. Query Precision

- Modality-specific processing enables precise responses
- Text-to-SQL conversion for structured data queries
- Vision-language integration for image understanding

3. Scalability

- Vector-based retrieval scales efficiently with document size
- Modular architecture allows independent component optimization
- Supports diverse document formats through Docling integration

4. Extensibility

- Plugin architecture for adding new content modalities
- Configurable model selection for different use cases
- API-compatible with existing RAG workflows

Disadvantages

1. Computational Overhead

- Multiple specialized models increase memory requirements
- AI-powered description generation adds processing latency
- GPU acceleration recommended for production deployment

2. Model Dependencies

- Requires multiple HuggingFace models for optimal performance
- Vision-language models are resource-intensive
- Quality depends on underlying model capabilities

3. Complexity

- More complex architecture compared to traditional RAG
- Requires understanding of multiple AI model types

- Additional debugging complexity for multi-modal failures

4. Storage Requirements

- Metadata preservation increases storage overhead
- Multiple model weights require significant disk space
- Temporary databases for table queries need management

Conclusions

This multi-modal RAG implementation addresses fundamental limitations in traditional document understanding systems by implementing semantic-aware parsing, modality-specific processing, and specialized retrieval mechanisms. The architecture successfully maintains information fidelity across heterogeneous document content while providing precise, context-aware responses to user queries.

The system demonstrates significant improvements in handling structured data, visual content, and code blocks compared to conventional text-only approaches. However, the increased computational complexity and resource requirements necessitate careful consideration of deployment constraints and use-case requirements.

Future enhancements could include support for additional modalities (audio, video), more sophisticated cross-modal reasoning, and optimized model architectures for reduced resource consumption. The modular design provides a solid foundation for these extensions while maintaining backward compatibility with existing RAG workflows.

The implementation serves as a practical example of how modern document AI can move beyond simple text processing to provide truly comprehensive document understanding and question-answering capabilities.

Retrieval Augmented Gen | Chatbots | Llamaindex | Docling | Artificial Intelligence