Ingestion Pipeline

This pipeline outlines the ingestion logic for a generative AI-powered research assistant built for a scientific journal publisher. The system is designed to:

- Detect newly uploaded journal files
- Chunk content into coherent segments
- Step 3: Build Metadata Per Chunk
- Post this step, I have already implemented a working prototype using Celery and Qdrant. that efficiently handles requests at scale (please refer to the upload endpoint)

Step 1: Detect Newly Uploaded Journal Files

```
 import boto3

def get_new_files_from_s3(bucket_name, prefix):
    s3 = boto3.client("s3")
    response = s3.list_objects_v2(Bucket=bucket_name, Prefix=prefix)
    new_files = []
    for obj in response.get("Contents", []):
        if not is_already_processed(obj["Key"]):
            new_files.append(obj["Key"])
    return new_files

for file_key in get_new_files_from_s3("journal-uploads-bucket", "daily/"):
    file_content = download_from_s3("journal-uploads-bucket", file_key)
    process_file(file_content)
```

Step 2: Chunk the Content into Coherent Segments

```
 # Pseudocode
def process_file(file):
    content = extract_text(file)
    # Use LangGraph's RecursiveCharacterTextSplitter with metadata-preserving features
    splitter = langgraph.RecursiveCharacterTextSplitter.from_defaults()
    chunks = splitter.split_text_with_metadata(content)
    for chunk in chunks:
        process_chunk(chunk, file_metadata)
```

Why LangGraph's Text Splitter?

LangGraph's `RecursiveCharacterTextSplitter` offers the following key strengths:

- **Contextual Coherence:** It respects sentence and paragraph boundaries, maintaining semantic flow in each chunk.
- **Customizable Chunk Size:** Supports adaptive chunk sizing while preserving important context.
- **Metadata Propagation:** Automatically maintains structural metadata (like headings or section IDs) useful for filtering and summarization.
- **Proven Compatibility:** Designed to work seamlessly with embedding models and vector stores, ensuring consistent chunking behavior for retrieval-augmented generation (RAG) systems

Step 3: Build Metadata Per Chunk

```
 # Pseudocode
def build_metadata(chunk, file_metadata):
    return {
        "id": generate_uuid(),
        "source_doc_id": file_metadata["doc_id"],
        "section_heading": chunk.section_heading,
        "journal": file_metadata["journal"],
        "publish_year": file_metadata["year"],
        "usage_count": 0,
        "attributes": extract_additional_metadata(chunk)
    }
```

Why Qdrant?

Qdrant is selected as the vector database because:

- It supports high-performance similarity search
- It allows rich metadata filtering
- It is optimized for scalability and production use
- Metadata filters like `source_doc_id`, `publish_year`, `journal`, or `attributes` can be used to precisely locate relevant chunks during semantic search

Importance of Metadata Filtering

Metadata filtering enables powerful, context-aware querying capabilities:

- Filter by specific journal or publication year
- Track and rank most referenced documents
- Exclude outdated or irrelevant segments
- Enable personalized or user-specific retrieval

This ensures the system provides well-grounded, high-quality responses backed by citations from authoritative journal content.

Advanced Considerations

Scalability of Embedded Models with Celery Background Tasks

Running embedded models as background tasks using Celery ensures scalability and efficiency:

- **Asynchronous Processing:** Celery allows embedding upsert tasks to run asynchronously, preventing bottlenecks in the main ingestion flow. For example, the embedding step can be offloaded from the main upload handler.
- **Distributed Workloads:** Celery's task queue distributes computation across multiple workers, enabling parallel processing of large batches of documents and chunks.
- **Resource Optimization:** Embedding models (e.g., for generating chunk embeddings) are lightweight compared to LLMs and can be efficiently executed in parallel. Celery's ability to schedule and prioritize tasks ensures efficient use of compute.
- **Fault Tolerance and Reliability:** Celery's retry mechanisms and task monitoring ensure robust handling of failures, making the system reliable even under heavy ingestion load.
- I would use **Celery background tasks** to upsert embeddings. Please refer to the tech documentation. I have already implemented a working prototype.
- As fields of research grow, separate **Qdrant collections** can be maintained per research domain (e.g., biology, materials science, AI) to improve retrieval relevance and scalability.
- Use of `upsert` operations ensures deduplication of embeddings while keeping document metadata fresh.