2025-06-25
Experiment for testing different Chunking strategies for various documents.

Obj: To systematically evaluate the effectiveness of various document chunking strategies for Retrieval-Augmented Generation (RAG) applications. This experiment will measure the impact of different chunking methods on retrieval and answer generation quality across diverse document types.

Key Performance Metrics:

Quantitative Metrics:

- Chunking Efficiency: Number of chunks generated and total ingestion time.
- Retrieval Speed: Time taken to retrieve relevant chunks for a given query.

Qualitative Metrics:

- Answer Quality: Assessed through impartial metrics like cosine similarity and automated evaluation frameworks (RAGAs and LlamaIndex). User rankings will also be incorporated to gauge perceived relevance.

The experiment will consist of many Tests. The first Test being on pdf. Then testing the pdf with different chunking strategies. Then we will do the same with docx and so on.

Independent Variable: The chunking strategy employed. A new strategy will be used for each test cycle.

Dependent Variables: Chunk count, ingestion/retrieval time, and answer quality scores.

Controls: To ensure a fair comparison, the following components will remain constant across all tests:

Data: A consistent set of 5-10 relevant documents of varying sizes will be used for each document type (e.g., PDF, DOCX).

Embedding Model: `bge-m3:567m`

Vector Database: Qdrant

Language Model (LLM): `llama3.1:8b`

Query Set: A standardized list of 3-5 questions will be used for each document type.

Chunking Parameters: Chunk size and overlap will be kept consistent initially to isolate the effect of the chunking strategy itself.

The Chunking Strategy is being tested. It will change after each test. We will use the llamaindexing pipeline with the following chunking strategies:

“SimpleDirectoryReader” paired with a loader like “smartPDFLoader” for locally loading files.

Once loaded Chunking will be done by:
- Node Parsers or “splitters”
`SentenceSplitter`, **`TokenTextSplitter`**, **`SemanticSplitterNodeParser`**, **and** **`RecursiveCharacterTextSplitter, HierarchicalNodeParser,`**

-Specialized Parsers: **`CodeSplitter, JSONNodeParser, HTMLNodeParser, and MarkdownNodeParser`**

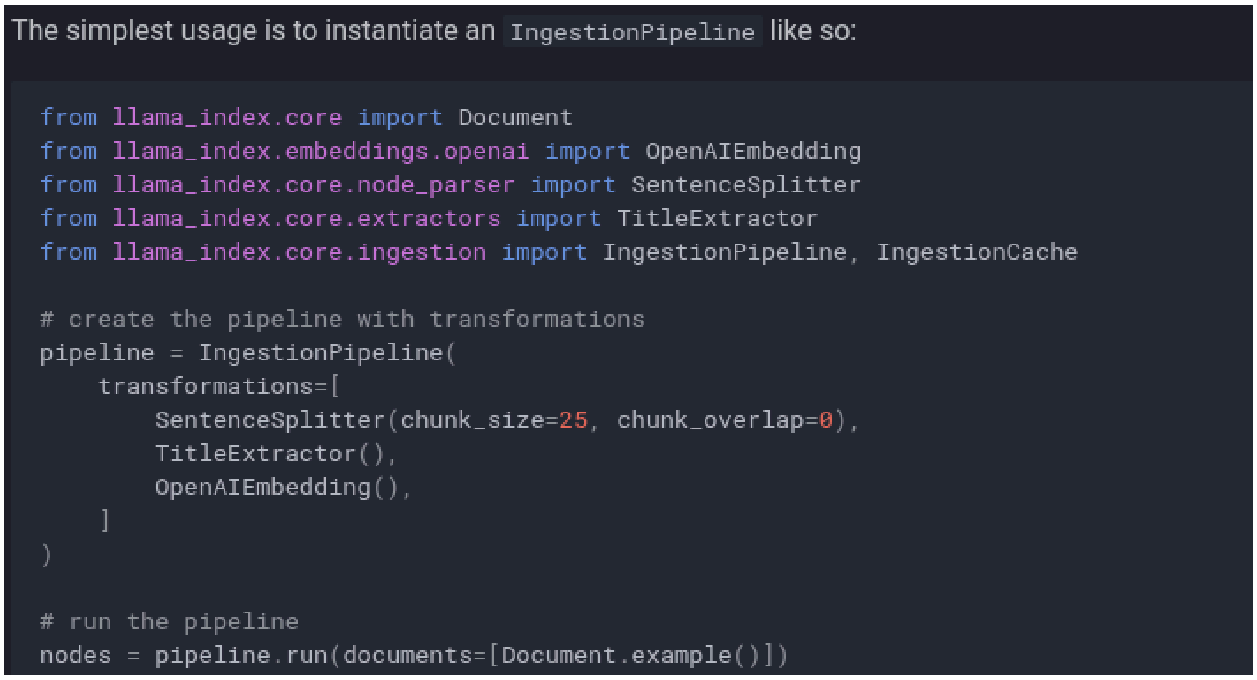
The normal Node Parsers will be tested on the same file types the Specialized parsers will only be used on their specific file types.

Steps:
Create empty Qdrant collections

Create file structure - directory for the documents that are raw, then a directory of the files that have been parsed/chunked and are ready to be embedded into Qdrant. This way we can come back eventually after finding the best chunking strategy and do the same test but with different embedding models.
Maybe something like this?:

```
rag_setup/Chunk_Test1.0$ tree
.
├── configs
│   ├── recursive_character.yaml
│   ├── semantic_splitter.yaml
│   ├── sentence_splitter.yaml
│   └── token_text_splitter.yaml
├── data
│   ├── processed_chunks
│   │   ├── processed_docx
│   │   └── processed_pdf
│   └── raw
│       ├── docx
│       └── pdf
├── docker-compose.yml
├── Dockerfile
├── ollama
├── Qdrant
├── requirements.txt
├── run.py
└── src
    ├── evaluate.py
    └── pipeline.py
```

Create llamaIndexPipeline:



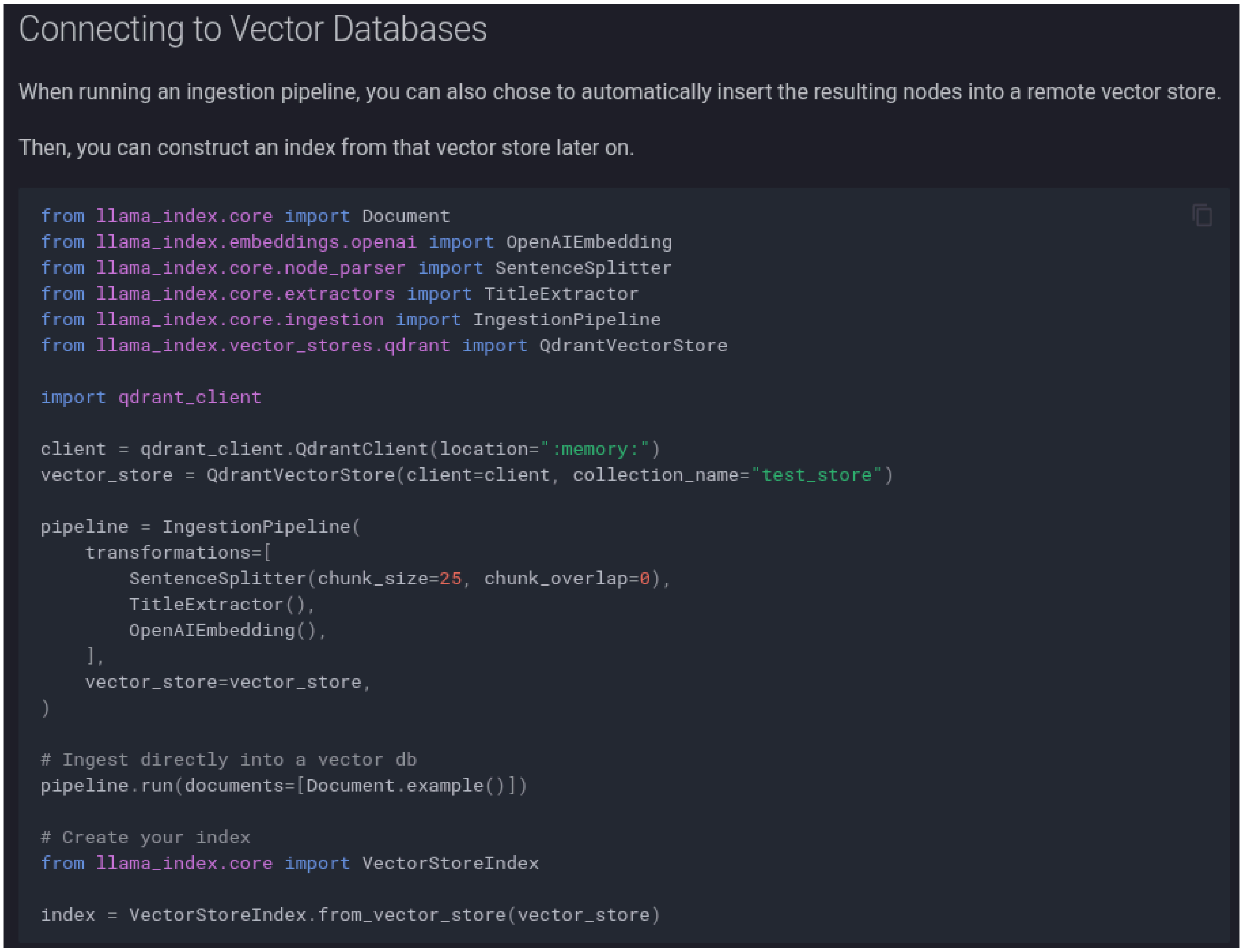
The simplest usage is to instantiate an `IngestionPipeline` like so:

```python
from llama_index.core import Document
from llama_index.embeddings.openai import OpenAIEmbedding
from llama_index.core.node_parser import SentenceSplitter
from llama_index.core.extractors import TitleExtractor
from llama_index.core.ingestion import IngestionPipeline, IngestionCache

# create the pipeline with transformations
pipeline = IngestionPipeline(
    transformations=[
        SentenceSplitter(chunk_size=25, chunk_overlap=0),
        TitleExtractor(),
        OpenAIEmbedding(),
    ]
)

# run the pipeline
nodes = pipeline.run(documents=[Document.example()])
```

The SimpleDirectoryReader would allow us to include local files.


Connecting to Vector Databases

When running an ingestion pipeline, you can also chose to automatically insert the resulting nodes into a remote vector store.

Then, you can construct an index from that vector store later on.

```
from llama_index.core import Document
from llama_index.embeddings.openai import OpenAIEmbedding
from llama_index.core.node_parser import SentenceSplitter
from llama_index.core.extractors import TitleExtractor
from llama_index.core.ingestion import IngestionPipeline
from llama_index.vector_stores.qdrant import QdrantVectorStore

import qdrant_client

client = qdrant_client.QdrantClient(location=":memory:")
vector_store = QdrantVectorStore(client=client, collection_name="test_store")

pipeline = IngestionPipeline(
    transformations=[
        SentenceSplitter(chunk_size=25, chunk_overlap=0),
        TitleExtractor(),
        OpenAIEmbedding(),
    ],
    vector_store=vector_store,
)

# Ingest directly into a vector db
pipeline.run(documents=[Document.example()])

# Create your index
from llama_index.core import VectorStoreIndex

index = VectorStoreIndex.from_vector_store(vector_store)
```

The above code shows how we would embed the document into Qdrant. But we want to add the document into Qdrant and also add the normal chunks into the “Chunked_data” directory. So we would add a “simpleFileWriter(output_dir=”processed_data/”) line after the OpenAIEmbedding() line. This way after the testing of the chunking method we can come back and get this chunk with a different embedding model.

We will use this with the SimpleDirectoryReader for local files but we will use Apify Actors for scratching sites, probably RSS.

Embed the chunks into Qdrant.
After embedding the chunks into Qdrant we will browse to Qdrant to see the collection sizes etc.

Query Qdrant and score the responses.

When conducting the experiment we should only have to delete the collections and then run something like this: python run.py --config configs/sentence_splitter.yaml.

This way we don’t have to rewrite code between tests.