WEEK 2 / DAY 8

LlamaIndex
Deep Dive

The Data Framework for Context-Augmented LLMs.

Today's Agenda

Part 1: Philosophy

LangChain vs. LlamaIndex.

Part 2: Core Architecture

Nodes, Indices, and Query Engines.

Part 3: Advanced RAG

Query Transformations & Re-ranking.

Part 4: Hands-On Lab

Building a Data Agent vs. LangChain.

LangChain vs. LlamaIndex

LangChain

"The Generalist"

Focuses on composability, chains, and agents. Great for glue code and multi-step logic.

LlamaIndex

"The Data Specialist"

Focuses on **Ingestion, Indexing, and Retrieval**. Optimized for RAG and handling massive datasets.

The "Data-First" Approach

LLMs are great, but they don't know **your** data.

LlamaIndex solves the "Data Connection" problem better than anyone else:

Data Loaders

LlamaHub has 100s of loaders (Notion, Slack, SQL, PDF).

Structuring

Organizing data into Graphs and Trees, not just flat lists.

Retrieval

Advanced algorithms to find the *exact* context needed.

1. Documents & Nodes

Document

A generic container for any data source (PDF page, API result).

Node

The atomic unit of data in LlamaIndex. A "chunk" of a source Document.

Nodes contain metadata (relationships to previous/next nodes).

```
from llama_index.core import Document
text_list = ["hello", "world"] documents =
[Document(text=t) for t in text_list]
```

2. The Index

An **Index** is a data structure composed of Nodes.

The most common is the **VectorStoreIndex**.

Documents → **Parse to Nodes** → **Embed** → **Vector Store**

Creating an Index

LlamaIndex abstracts away the embedding and storage complexity.

```
from llama_index.core import VectorStoreIndex # One line to embed and index everything  index =
VectorStoreIndex.from_documents(documents)
```

3. The Query Engine

A **Query Engine** is a generic interface that allows you to ask questions over your data.

It takes a natural language query and returns a rich response.

```
query_engine = index.as_query_engine()
response = query_engine.query(
"What is the author's name?"
) print(response)
```

What is a "Node" in LlamaIndex?

A. A server in a cluster.

B. A chunk of data from a source Document with metadata.

C. The embedding model itself.

D. A prompt template.

Moving Beyond Basic RAG

Basic RAG (Top-K Similarity Search) fails when:

- The user asks a complex multi-part question.
- The answer requires summarizing the entire document.
- The relevant context is buried deep in the document.

LlamaIndex specializes in fixing these issues.

Technique 1: Query Transformations

Sometimes the user's query is bad. We use the LLM to rewrite it.

HyDE (Hypothetical Document Embeddings): The LLM hallucinates a hypothetical answer, and we use *that* for the vector search.

Reasoning

A hypothetical answer usually looks closer to the real document in vector space than a short question does.

Technique 2: Routing

If you have different data sources (e.g., "Sales SQL DB" and "HR Handbook PDF").

A **RouterQueryEngine** uses the LLM to decide which index to query.

```
# Concept Code tools = [ QueryEngineTool(engine=sales_engine, name="sales"),
QueryEngineTool(engine=hr_engine, name="hr") ] router =
RouterQueryEngine(selector=LLMSingleSelector(...), tools=tools)
```

Technique 3: Re-Ranking

The Problem: Vector search returns the "closest" matches, but they might not be the most relevant for the specific answer.

The Solution: Retrieve 50 nodes, then use a specialized "Re-Ranker" model (like Cohere) to sort them by relevance and keep the top 5.

Recall vs. Precision

Retrieve broad (High Recall), then Filter down (High Precision).

What is the purpose of Re-Ranking in RAG?

A. To compress the vector database size.

B. To re-order retrieved nodes by relevance before sending to the LLM.

C. To translate the query into SQL.

D. To embed the document again.

Hands-On Lab Overview

We will build a robust RAG system using LlamaIndex.

- **Input:** The same PDF from Day 7 (to compare frameworks).
- **Engine:** VectorStoreIndex.
- **Model:** OpenAI GPT-4o.
- **Advanced Feature:** We will implement a persistent index storage.

Step 1: Installation

LlamaIndex is modular. We need the core and specific integrations.

```
%pip install llama-index llama-index-llms-openai %pip install llama-index-embeddings-openai
```

Step 2: Global Settings

Unlike LangChain where you pass the model to every chain, LlamaIndex uses a global Settings object (though you can override it).

```
import os from llama_index.core import Settings from llama_index.llms.openai import OpenAI
os.environ["OPENAI_API_KEY"] = "sk-..." # Set global defaults Settings.llm = OpenAI(model="gpt-4o",
temperature=0)
```

Step 3: Loading Data

SimpleDirectoryReader is the magic function.

It automatically detects file types in a folder (PDFs, TXTs, MDs) and parses them into Documents.

```
from llama_index.core import
SimpleDirectoryReader
# Reads all files in the 'data' folder
documents =
SimpleDirectoryReader("./data").load_data()
print(f"Loaded {len(documents)} docs" )
```

Step 4: Indexing

This single line does 3 things:

1. Splits documents into Nodes (Chunking).
2. Calls OpenAI to create Embeddings for each Node.
3. Stores them in an in-memory Vector Store.

```
from llama_index.core import VectorStoreIndex index = VectorStoreIndex.from_documents(documents)
```

Step 5: The Query Engine

```
query_engine = index.as_query_engine() response = query_engine.query(
"What is the attendance policy?"
) print(response)
```

By default, this performs a top-k similarity search and synthesis.

Storage Context

Creating embeddings costs money. Don't do it every time you run the script.

Save the index to disk!

```
# Save
index.storage_context.persist(persist_dir="./storage")
# Load from llama_index.core import StorageContext,
load_index_from_storage storage_context =
StorageContext.from_defaults(persist_dir="./storage")
index = load_index_from_storage(storage_context)
```

Comparison: LangChain vs LlamaIndex

Feature	LangChain (Day 7)	LlamaIndex (Day 8)
Ingestion	Manual Loaders/Splitters	SimpleDirectoryReader (Magic)
Complexity	High Control / Verbose	High Abstraction / Concise
Best For	General AI Apps / Agents	Search / Retrieval / RAG

Which function is used to automatically parse a folder of files in LlamaIndex?

A. FolderLoader

B. PyPDFLoader

C. SimpleDirectoryReader

D. VectorStoreIndex

Activity: The "Index" Game

Scenario: You have a 1000-page textbook.

Challenge: Describe how you would organize this data for an LLM to answer "Summarize Chapter 5".

- **Vector Search?** Might miss the overall theme.
- **Keyword Search?** Too noisy.
- **LlamaIndex Tree Index?** Perfect for summarization (hierarchical).

Not Just Vectors

Summary Index

Good for... summarizing documents.

Tree Index

Good for hierarchical traversal of information.

Keyword Table

Good for exact routing based on keywords.

Metadata Extraction

LlamaIndex can automatically extract metadata (Title, Date, Authors) from nodes before indexing.

This allows for **Metadata Filtering** during retrieval.

```
# Example Filter  filters = MetadataFilters( filters=[ExactMatchFilter(key="year", value="2023")] )
```

Chat Engine

Query Engine: Stateless. Ask a question, get an answer.

Chat Engine: Stateful. Remembers the conversation history (Memory).

```
chat_engine = index.as_chat_engine()
response = chat_engine.chat(
"Tell me about the author."
) response =
chat_engine.chat("What else did he write?" )
```

Day 8 Summary

- **LlamaIndex** is the go-to framework for RAG and Data Agents.
- **Nodes** are the fundamental unit of data.
- **Indices** structure these nodes for retrieval.
- We learned to load, index, persist, and query data with just a few lines of code.

Looking Ahead: Day 9

Multi-Agent Systems (CrewAI)

Moving from a single RAG agent to a team of agents.

Q & A

Open floor for questions on RAG, Vector DBs, or LlamaIndex.

Deep Dive: Node Parsing

LlamaIndex offers advanced parsing:

- **SentenceWindowNodeParser:** Keeps a "window" of surrounding sentences for context.
- **HierarchicalNodeParser:** Creates parent/child relationships for chunks.

Deep Dive: Response Synthesis

How does the engine generate the final answer?

- **Refine:** Iteratively updates the answer with each retrieved chunk.
- **Compact:** Stuffs as many chunks as possible into the context window.
- **Tree Summarize:** Summarizes chunks recursively.

Buffer Slide: Discussion on Data Privacy

Buffer Slide: Troubleshooting Installation Issues

Buffer Slide: Capstone Project Alignment

Buffer Slide: Review of Day 7 Code

Buffer Slide: Advanced Metadata Filtering

Buffer Slide: Using LlamaParse

Buffer Slide: Multimodal RAG (Images)

See you tomorrow!

Prepare your Python environments for CrewAI.