Over the past three days, we laid the groundwork for building AllyIn Compass, a smart enterprise assistant capable of handling both structured and unstructured data. The journey began with setting up a clean development environment. I created a Conda Python environment, installed essential tools like Docker, VS Code, and the required Python libraries (pandas, duckdb, etc.), and initialized the project directory with a proper folder structure to organize code and data.

For Day 1, we focused on ingesting structured data. I generated three sample .csv files—customers.csv, orders.csv, and emissions.csv—to simulate enterprise datasets. Using Python and the DuckDB engine, I wrote a script (structured\_loader.py) that automatically loads all CSVs into DuckDB as SQL tables. This allowed us to simulate working with real business records in a local SQL environment, verifying everything by querying the tables and previewing the data.

On Day 2, the focus shifted to unstructured data such as PDFs and emails. I created realistic test documents: three PDFs representing reports and memos, and three .eml files containing typical email communication. Using PyMuPDF and Python’s built-in email library, I extracted the text content from each document and saved the parsed results into a unified JSONL file (parsed.jsonl). This file structured all the unstructured content into a standard format, ready for downstream processing.

Day 3 was dedicated to turning that raw text into machine-understandable vector embeddings. I used the sentence-transformers library to convert the extracted text into dense numerical vectors, which capture the semantic meaning of each document. These embeddings were then stored in Qdrant, a vector database running in Docker. I used the embedder.py script to upload all six documents as vector points to the docs collection in Qdrant, each tagged with its source file name. To verify the setup, I also wrote a simple script (vector\_inspect.py) to fetch and print the stored vectors along with snippets of their content, confirming that everything was correctly indexed and ready for semantic search.

On Day 4, I set up three parallel retrieval systems to support different types of enterprise queries:

• **SQL Retriever:** Initially used SQLDatabaseChain from langchain\_experimental.sql to connect DuckDB with OpenRouter’s mistral-7b-instruct LLM for SQL generation. Later transitioned to create\_sql\_query\_chain from langchain.chains.sql\_database.query, which enabled better SQL extraction, cleaning, and manual execution for reliability.

* **Vector Retriever**: I built a custom semantic search function that queried Qdrant for the top-k similar documents based on input queries. This enabled answering questions using email and PDF content.
* **Graph Retriever**: I installed Neo4j and created a simple knowledge graph with 10 entities (e.g., Plant A, CO2 Limit) and relationships (EXCEEDS, COMPLIES\_WITH, etc.). Using Cypher queries via the official Neo4j driver, I built graph\_retriever.py to fetch and return regulatory relationships dynamically.
* Day 5 was all about **intelligence orchestration**. I created reusable tools (Tool(...)) for each retriever: SQLTool, VectorTool, and GraphTool. Then, using LangChain’s initialize\_agent, I built a **multi-tool agent** that could dynamically choose which retriever to use based on user input. This enabled chained reasoning like:
* *“What regulation does Plant A exceed and what products were ordered?”*
* The agent correctly used the GraphTool to find that *Plant A exceeds CO2 Limit*, then used SQLTool to look up the ordered product (*Laptop*) based on facility ID. I also refined the agent’s tool descriptions and adjusted SQL logic to avoid ambiguous joins or schema mismatches. Logging and error handling were added to trace each tool’s usage with timestamps and input/output.