Mini Unified Data Intelligence Engine

10-Day Beginner-Friendly Enterprise Project

Goal: Build a lightweight, end-to-end prototype of a smart enterprise data system. It will load spreadsheets and PDFs, understand text using AI, store everything in smart databases, and answer user questions clearly using Retrieval-Augmented Generation (RAG). It will include feedback logging, basic graphs, and a visual dashboard — all in 10 days, suitable for early-career developers.

👩‍💻 Who Can Do This

- Junior-level undergrad with basic Python knowledge
- No prior LangChain or Neo4j experience required
- Some knowledge of JSON, SQL, and REST APIs helpful

🔧 Tech Stack Summary

- Python (main language)
- DuckDB (for structured data)
- Qdrant (for semantic search)
- Neo4j (for knowledge graph)
- Streamlit (for UI)
- LangChain (for chaining LLMs and tools)
- OpenAI or Fireworks.ai (for answering queries)

📁 Folder Layout

```
Unset
project/
├── ingest/       # data loaders
├── retrievers/   # SQL, vector, graph tools
├── tools/        # RAG, agent, feedback
├── ui/           # Streamlit app
```

```
├── feedback/     # thumbs up/down logger
├── config/       # constants, routing rules
└── README.md
```

🗓️ Day-by-Day Plan

Day 1: Structured Data Ingestion Setup

Goal: Load spreadsheets (.csv, Google Sheets) into DuckDB.

Step-by-Step:

- Set up Python env: `python -m venv venv && source venv/bin/activate`
- Install DuckDB: `pip install duckdb pandas`
- Ingest sample CSV files:

```
Unset
import pandas as pd
import duckdb
customers = pd.read_csv('data/customers.csv')
duckdb.sql("CREATE TABLE customers AS SELECT * FROM customers")
```

- Save normalized tables: `staging_customers`, `staging_orders`

Deliverables:

- `ingest/load_structured.py`
- DuckDB working with 2 tables

Day 2: Ingest Unstructured Data (PDF + Email)

Goal: Parse PDFs and .eml emails into text with metadata.

Step-by-Step:

- Use PyMuPDF for PDFs:

```
Unset
import fitz
with fitz.open("sample.pdf") as f:
    text = "".join(page.get_text() for page in f)
```

- Use Python `email.parser` for .eml
- Store data as:

```
Unset
{"doc_id": "xyz", "source_type": "email", "title": "...", "body":
"...", "created_at": "2024-12-01"}
```

Deliverables:

- `ingest/parse_unstructured.py`
- `parsed_docs.jsonl`

Day 3: Embedding + Metadata-Aware Indexing

Goal: Turn docs into searchable AI vectors with metadata.

Step-by-Step:

- Install sentence-transformers: `pip install sentence-transformers`
- Chunk long text:

```
Unset
from langchain.text_splitter import
RecursiveCharacterTextSplitter
chunks =
RecursiveCharacterTextSplitter(chunk_size=500).split_documents(do
cs)
```

- Generate embeddings, store in Qdrant

Deliverables:

- `tools/embedding.py`
- Indexed Qdrant collection with metadata

Day 4: Hybrid Semantic + SQL Retrieval

Goal: Set up a query system using both DuckDB and Qdrant.

Step-by-Step:

- Create SQL retriever using LangChain
- Create vector retriever using LangChain VectorStoreRetriever
- Implement a simple fallback: vector → SQL → default answer

Deliverables:

- `retrievers/sql.py`, `retrievers/vector.py`
- `tools/router.py`
- Test notebook with sample queries

Day 5: Knowledge Graph & Entity Extraction

Goal: Add a small graph database of named entities.

Step-by-Step:

- Install Neo4j Desktop or use sandbox
- Extract entities using spaCy:

```
Unset
import spacy
nlp = spacy.load("en_core_web_sm")
doc = nlp("Microsoft acquired OpenAI")
```

- In Neo4j, define:
  - Nodes: Document, Person, Company
  - Relations: MENTIONS, WORKS_AT

Deliverables:

- `retrievers/graph.py`
- Mini graph with 10+ nodes

Day 6: LangChain Agent with Tool Routing

Goal: Automatically select the right retriever using LangChain agent.

Step-by-Step:

- Register tools: SQLTool, VectorTool, GraphTool
- Create an `AgentExecutor` with routing logic
- Add trace logging: log input, tool used, output

Deliverables:

- `agents/router_agent.py`
- Agent can answer "Which customer ordered most in Jan?"

Day 7: RAG Interface & UI Prototyping

Goal: Create a basic UI + plug in RAG answering.

Step-by-Step:

- Use Streamlit:

```
Unset
import streamlit as st
st.title("Enterprise AI Search")
query = st.text_input("Ask something")
```

- Show results with citations
- Highlight which tool was used (SQL, vector, graph)

Deliverables:

- `ui/app.py`

- Running UI with 3 working queries

Day 8: Feedback & Relevance Logging

Goal: Let user rate results and store logs for retraining.

Step-by-Step:

- Add thumbs up/down
- Save feedback as JSON:

```
Unset
{"query": "...", "doc_id": "abc", "rating": 1, "comment": "good"}
```

- Visualize ratings over time with bar chart

Deliverables:

- `feedback/logger.py`
- `feedback/feedback_log.jsonl`

Day 9: Observability + Query Auditing

Goal: Add simple metrics and logs to monitor system behavior.

Step-by-Step:

- Track:
  - Query count per day
  - Which tool was used
  - Average query time
- Show dashboard in Streamlit sidebar

Deliverables:

- `dashboards/metrics.py`
- Screenshot of 3 metrics

Day 10: Final Polish + Simulated Fine-Tuning

Goal: Clean up code, document system, and simulate model improvement.

Step-by-Step:

- Organize folders, add `README.md`
- Record a 2–3 min demo using Loom
- Filter positive feedbacks and run `simulate_finetuning.py`

Deliverables:

- Final `README.md`
- GitHub repo with full working system
- Demo video showing UI + answering

✅ Optional Stretch Goals (Pick 1)

- Add OCR for scanned PDFs
- Use ColBERT for reranking answers
- Color-code recent vs old docs in results
- Add tool confidence scores and thresholds