Technical Challenge: Data Engineer

Objective

Develop and implement a system that leverages an ETL pipeline to provide fast searching capabilities for domain entities. The data should follow a data lake architecture, enabling seeding the data and adding new data, both structured and unstructured. Then the data should be refined in a data warehouse following a star schema in combination with medallion architecture. For example, using star for the silver and gold layer and leaving the bronze layer with structured / unstructured data.

The system should provide fast searching capabilities by leveraging a Vector DB, its entities may match with one of the entities of the relational database.

The system should expose a REST API that enables:

- Seeding data at the beginning.
- Full CRUD of raw data.
- Querying Gold Layer.
- Searching by leveraging the Vector DB.

Technical Details

All the code should be hosted and version controlled in **Github as a public repo.** The data lake should use an open-source SQL solution (e.g. SQLite / Delta Lake / Parquet Files) and the Vector DB should be Typesense. The REST API should be developed with FastAPI.

The structure of the code should follow a Layered-Architecture, with at least the following layers:

1. API/Routes
2. Controller / Services
3. Domain Entities
4. Data Access

The Domain entities should not be the same as the entities in the Data Access layer.

The REST API should create the necessary schemas if they do not exist. The input for the system should include both structured and unstructured data, e.g. JSON files, CSV files and PDF files.

Source data

The candidate is free to use any data they choose as long as it is publicly available and sufficiently complex to support the use case given. It is also possible to use tools like SVD to generate synthetic data.

Note on AI tools

The usage of GenAI tools (e.g., ChatGPT, Cursor, TabNine, Github Copilot) is allowed and it is expected to streamline repetitive and/or easy to automate processes such as the following:

- Generate descriptive git commit messages.
- Generate docstrings for functions.
- Generate drafts for tests and boilerplate code.
- Use as a peer review tool to identify potential flaws.
- Identify which design patters will be most suitable for a particular functionality
- Feel free to experiment and leverage these tools to enhance your work.

Mandatory Features

- Create Mermaid diagrams for the data structure
  - One Diagram per layer in the medallion architecture.
  - One Diagram for the overall data lake organization.
- SQL Tables:
  - The ETL should be done with a framework (pandas / polars / pyspark). Not by running SQL queries.
  - The SQL DB should be accessible (either using DB files or credentials).
  - The Models should be defined with an ORM (e.g. SQLModel)
  - The data warehouse should have:
    - At least one fact table.
    - At least five dimension tables.
- REST API:
  - The Search endpoint should use at least one filter.
  - The CRUD should allow for single instances and batch operations.
  - The API should implement authentication. E.g. using Basic Auth
- Vector DB:

It should be run through Docker/Docker Compose.

- Overall:

  All the code should have type hints.

  Domain Entities should be defined with Pydantic.

Optional Features (include at least 1):

The following optional features are divided by topic, they can be completed in any combination or order.

Database Related

1. Use a Calendar table.
2. Use Materialized Views to support specific queries.
3. Use a hosted DB. e.g. Supabase.
4. Use PySpark 3 for the ETL Transformation.
5. Use at least one user defined function for a non-trivial transformation.
6. Support Database migrations. E.g. using Alembic.
7. Use DVC for the data used in the seeding process.
8. Include control of data governance / data lineage.

API Related

9. The REST API endpoints are protected with Authentication/Authorization using JWTs.
10. Add a Data Mart to the database and create a dedicated Router to consume those tables.
11. The API implements the Async Request-Reply pattern.
12. Expose an GraphQL endpoint.
13. Use versioned APIs. I.e. create a v1 and a v2 version of the same router with different implementations.

Business Related

14. Enrich the raw data by consuming an external API
15. Use some orchestration tool to automatically ingest data whenever the user drops a file into a folder. I.e. by using Airflow / Dagster or similar with a File Trigger.
16. The system works with a PubSub pattern (communication based on Queues). E.g. Kafka, redislite+Celery, redislite+rq, huey (with redislite, sqlite o in-memory). Redislite+dramatiq.

ETL Related

17. Use the Builder Pattern.
18. Use the Chain of Responsibility Pattern.
19. Use the Repository Pattern.

Software Engineering Related

20. Lock your dependencies. e.g. uv, pipenv, poetry or pdm.
21. Use Dependency injections. E.g. by using inject or dependency injector
22. The application leveraging a logging library. E.g. built-in logging or loguru
23. The system should be packaged in a wheel file with an entrypoint.
24. Make a single docker compose with the necessary services.
25. Add a linting with Pylint and Ruff.
26. Add static type checking with Pandera and Mypy.
27. Add a Dev Container configuration.
28. Add unit tests using pytest.
29. Add integration tests using pytest and test client from FastAPI and polyfactory.

Evaluation criteria

The project will be evaluated in four dimensions:

1. **Database design**: how the Data Warehouse is structured, its data types, relationships, etc.
2. **Best practices**: a code review to evaluate best practices, patterns and overall design.
3. **Complexity**: the level of optional features implemented and how they were incorporated into the development process.
4. **Organization**: the most important aspect is to have something functional by the deadline, the candidate should be able to organize themselves and deliver something valuable.
5. **Innovation**: new features outside of what has been asked is a plus.

Helpful Resources

These resources may help you get started:

- Request-Reply Pattern:
  - Microsoft Architecture Patterns
- Medallion Architecture
  - Databricks Glossary
  - Microsoft Lakehouse Documentation

- SQLite:
  - Creating Database
- Supabase:
  - Initializing
  - YouTube Tutorial
- SQLAlchemy ORM:
  - YouTube Tutorial
- Typesense:
  - Getting Started