31 october 2025 Deadline Report

Francesco Pivotto 2158296
Giorgia Amato 2159999
Alessio Demo 2142885

31 ottobre 2025

Indice

1 Introduction

In this part of our project, we deal with the database containing all the data that we need for our purpose. In details we need to establish a connection to connect the database with the data resources, that allows us to create an "running instance" of the database which we can use to query the database itself and store the results returned, in particular in our case the results will be stored in a custom JSON format called 'Simple JSON format' provided by the tutor. Next we will carry out the evaluation of the results produced by our system using the ground truth data as comparator.
Let's go into details.

2 Libraries & Components

2.1 System Architecture: Dependency Stack and Key Requirements:

The project is built upon a Python dependency stack prioritizing robustness, efficient data analysis, and flexible configuration. All essential libraries are listed in requirements.txt.

1. Core Frameworks & Data Analysis (DataFrames): **pandas** and **numpy** are used for the efficient manipulation and analysis of tabular data structures (DataFrames) and complex numerical calculations.

2. Database and SQL: **duckdb**: Utilized as a high-performance, in-process analytical database, ideal for rapid processing of data volumes directly within the application. **sqlglot**: Employed for the abstract analysis, manipulation, and translation of SQL queries.

3. LLM Orchestrator (Large Language Model) We have implemented a streamlined pipeline to interact with Google's Gemini Large Language Model (LLM) using the **LangChain** framework and with IBM watsonx ai using the **ModelInference** package from **ibm_watsonx_ai.foundation_models**.
   LangChain: The core framework managing interactions with the AI model.
   Going deeper in the technical stuff fro Google Gemini LLM:

   - API Key Management Securely loads API keys from environment variables using dotenv, ensuring flexible and safe configuration.
   - LLM Initialization Instantiates a ChatGoogleGenerativeAI object with customizable parameters such as model version (gemini-2.5-flash) and temperature.
   - Model Warm-Up Performs a quick direct prompt to initialize the model and measure response latency.
   - LangChain Prompt Chaining Uses a structured prompt template in Italian to generate clear and concise answers. The chain includes:
     - ChatPromptTemplate: dynamically formats the prompt
     - ChatGoogleGenerativeAI: generates the response
     - StrOutputParser: extracts the textual output

   Then regarding IBM watsonx ai LLM:

- API Key Management Securely loads API keys from environment variables using dotenv, ensuring flexible and safe configuration.
- The code builds a ModelInference object configured with the credentials, project ID, and chosen model.
- The system submits the prompt and receives the response from the LLM.
- Logging the prompt, response length, latency, and any errors (including Python stack traces if exceptions occur)

In summary he script sends a sample prompt and logs the LLM's output.

4. Configuration and Data Validation: **pydantic**: Essential for data validation and structured configuration management. It enforces type safety, handles data parsing/coercion, defines structured nested models, and raises immediate errors on malformed input. **python-dotenv**: Loads environment variables (including secrets) from the .env file. **pyyaml**: Handles reading and writing configuration files in YAML format.

5. Templating: **jinja2**: The templating engine that will be used for the dynamic generation of structured output, such as code, reports (HTML, XML, JSON), or complex prompt statements, by separating presentation logic from data.

6. Logging and Utilities **loguru**: We implement structured logging with `loguru` instead of the basic Python `logging` module. The goal is to have uniform, timestamped messages with contextual metadata (dataset, query, latency, token usage), visible in the console and saved to rotating log files. Specifically, when the code is executed, a "logs" folder will be generated and inside it there will be a text file containing all the system logs.

   - **Structured and unified output:** provides a single, centralized configuration for message format, log levels, and output handlers.
   - **Enhanced observability:** captures execution timings (in milliseconds) for DuckDB `EXPLAIN`/`EXPLAIN ANALYZE` operations and LLM requests, including token usage statistics when available.
   - **Data persistence and maintenance:** ensures automatic log file rotation, retention, and compression to manage disk usage and prevent oversized log artifacts.

   We will see later in the report more details about the Logs handling.

7. Development and Testing **pytest**: The standard testing framework for executing all tests. coverage and **pytest-cov**: Used to measure the percentage of source code covered by automated tests (Code Coverage). **pytest-mock**: Facilitates the creation of mock objects and stubs to isolate external dependencies during unit tests.

3 Project Structure

The project follows a modular organization designed to separate configuration, data, source code, and testing components. This layout enhances maintainability, scalability, and code clarity.

Before seeing the project's structure, we want to highlight the different design patterns used: The **src** folder is divided into logical packages (db, llm, utils, test, etc.), that highlights the separation of concerns, provides a easier navigation and a modular code organization.

Furthermore, within the `src/llm` directory, the **Adapter Pattern** will be implemented through the introduction of a dedicated **Adapter class**. Its primary purpose will be to manage the interface between our core system and the various Large Language Model Providers.This approach ensures that the core system can communicate with all models *uniformly* and *decoupled*, effectively shielding the code from future variations in individual vendor APIs.

In our project we applied also a **Repository Pattern** by means a single script for connecting the database e instantiating it within each dataset folder. Finally the main script provides a single interface that hides the complexity and the logic of underlying functionalities (db connection, sql queries handling, LLM interaction ...).

3.1 Main Directories and Contents

Directory	Description and Contents
/**config**	Contains configuration files and the logic for their loading. Includes: **config.yaml** — Main configuration file. **loaders.py** — Parses the YAML configuration into a Python structure.
/**data**	Stores raw data, ground truth, and intermediate results. Contains: /**.ground_truth** — Baseline truth for evaluation. /**.output** — Generated results ready for comparison.
/**results**	Holds the final outputs produced by the analysis process. Typically includes `.json` and `.txt` files from the `explain` and `analyze` operations.
/**src**	Main source folder containing the project's operational logic. Includes: **main.py** — Entry point for execution. /**db** — Database interaction modules (e.g., `db_connection.py`, `run_queries_to_json.py`). /**llm** — Interfaces with Large Language Models (e.g., `google_genai_connection.py`, `openai_connection.py`). /**galois** — Implements query optimization and execution logic inspired by the Galois architecture. /**utils** — Utility scripts: `constants.py`, `logging_config.py`, `build_ground_truth.py`, `galois_eval.py`.
/**test**	Contains unit and integration testing scripts (e.g., `db_utils.py`).

Tabella 1: Main directories and their contents.

3.2 Root-Level Files

File	Description
README.md	High-level project documentation.
requirements.txt	List of required Python dependencies.
setup_project.py	Script for project initialization.

Tabella 2: Root-level files.

4 Implementation Details and Project Setup

4.1 Project Setup and Launch

To launch the system, two fundamental scripts are required: **setup_project.py** and /**src/main.py**.

4.1.1 Initial Configuration and Dependencies

It is **mandatory** to execute `setup_project.py` the first time the system is launched or whenever dependencies are updated.

- `setup_project.py`**:** This script manages the environment configuration: it creates the virtual environment (`.venv`) and installs all necessary Python libraries, as specified in `requirements.txt`.
- `/src/main.py`**:** This module contains the main logic to run all system functionalities, from data processing to model execution.

4.1.2 Execution Modes

The system execution can be performed sequentially. Commands can be concatenated on a single line for quick launch:

Linux/macOS (Bash/Zsh)

```
$ python setup_project.py ; .venv/bin/python -m src.main [DATASETS]
```

Windows (CMD/PowerShell)

```
> python setup_project.py && .venv\Scripts\python -m src.main [DATASETS]
```

4.2 Configuration Handling

All system configuration details are managed via a dedicated YAML file, and the `config/loaders.py` module serves as the central hub for all configuration data. It handles the critical process of loading and validating the external YAML file, ensuring the application always runs with correct and predictable parameters.

4.2.1 The config/loaders.py Configuration Hub

The `config/loaders.py` module orchestrates the configuration life cycle through three main stages:

1. **Parsing (YAML):** The script uses the `PyYAML` library to read and parse the content of the `config/config.yaml` file into a standard Python dictionary.
2. **Structured Validation (Pydantic):** The parsed data is immediately processed and validated using the **Pydantic** library.
    - **Rigid Structure:** The configuration schema is defined by precise, nested classes (such as `DatasetConfig`, `IOConfig`, `ExecutionConfig`, etc.) that mirror the expected structure of the YAML file. The main class, `AppConfig`, aggregates all these sections.
    - **Type Safety:** Pydantic strictly enforces data types (e.g., ensuring `temperature` is a `float` and `max_retries` is an `int`). This eliminates common runtime errors caused by misconfigured settings.
    - **Predictability:** By validating the structure and types upfront, we guarantee that all required fields are present and correctly formatted, making the configuration predictable and reliable.
3. **Global Access:** The validated configuration is loaded by the system through a dedicated utility class. This allows any module in the project to access configuration details easily and consistently by calling `Config_Loader().getConfig()` , rather than relying on a direct global instance. For instance, a specific setting can be retrieved via `Config_Loader().getConfig().gemini.temperature`.

4.2.2 Dataset Override and Fallback Logic

The system employs a clear priority logic to determine which datasets to process, ensuring flexibility in execution:

1. **Command Line Override (Highest Priority):** Datasets specified as command line arguments to `main.py` take the highest priority, overriding any existing configuration.
2. **YAML Configuration:** If no command line arguments are provided, the system falls back to the list of datasets specified within the `config/config.yaml` file.
3. **Default Behavior (Fallback):** In the absence of both command line arguments and an explicit YAML configuration entry, the script automatically defaults to processing **all** available datasets in the project, guaranteeing system operability.

4.3 Database-Dataset Connection and Management

For our purpose we will use DuckDB as database manager, we believe that it is the ideal for our project since it is very powerful and rapid in the data processing within the system, moreover duckDB is an embedded database manager and can run within the instance of our system instead of establishing a server connection like postgres. At more technical level the **db_connection.py** connects to a DuckDB database file associated with a specific dataset. The next step is **duckdb_db_graphdb.py** which allows us to automatically build DuckDB databases from a directory of datasets, each containing an SQL ingestion script that creates the data tables and populates them, this allows us to have an instance of the db representing a particular dataset in which we can run queries.

4.4 Query Execution

The queries can be execute by means **run_queries_to_json.py** script that automates the execution of SQL queries on DuckDB database and exports their results to JSON files, one per query.
Going more deep the script executes each query and saves the results as JSON:

1. Iterates through each query tuple.
2. Executes the SQL query using the provided DuckDB connection.
3. Fetches the resulting rows and column names.
4. Converts the result into a list of dictionaries.
5. Saves each query result as a JSON file in the output directory.

4.4.1 Explain Plans (DuckDB)

The `EXPLAIN` module was implemented to validate the SQL execution pipeline on DuckDB and to visualize the difference between logical and physical plans, providing a reliable baseline for the comparison with the GALOIS framework.

Implementation Overview The functionality is implemented in the main script `src/db/run_explain_plans.py`, which delegates the actual DuckDB interaction to the adapter `src/db/duckdb\_explain.py`. The script can be executed by passing one or more dataset names (e.g., `world`) or the keyword `all` to process them all together. For each dataset located in `<repo-root>/data/<dataset>`, the corresponding database `<dataset>.duckdb` and all `queries_*.sql` files are loaded, split into single statements, and both `EXPLAIN` and `EXPLAIN ANALYZE` are executed.

The module's core responsibilities are:

1. **Dataset iteration:** enumerates datasets and SQL files and assigns a stable query identifier (e.g., `queries\_world\_\_q3`);
2. **Parsing:** splits multi-statement SQL files on semicolons while preserving comments for traceability;
3. **Execution:** connects to the dataset's `.duckdb` database and issues both `EXPLAIN` and `EXPLAIN ANALYZE`;
4. **Persistence:** stores all outputs through the `save_both()` function.

Output Structure The results are written under the global `results/` directory in two parallel trees:

- `explain_result/` for text files (`.txt`) for human inspection;
- `explain_result_json/` equivalent JSON files for machine processing.

Each of these folders contains one subdirectory per dataset (`flight-2`, `flight-4`, `fortune`, `geo`, `premier`, `presidents`, `world`). For every SQL statement, four output files are produced sharing the same base name (e.g., `queries_world__q3`):

- `__explain.txt` and `__explain.json` for logical and physical plans;
- `__analyze.txt` and `__analyze.json` for runtime statistics and operator timings.

Example A representative example from the `world` dataset is:

```
SELECT count(DISTINCT government_form)
FROM target.country
WHERE continent = 'Africa';
```

This query counts the number of distinct government forms for African countries. The generated logical plan follows a typical structure (*Aggregate* → *Filter* → *TableScan*), while the physical plan contains the corresponding operators (*HASH_AGGREGATE* → *FILTER* → *SEQ_SCAN*). The analyzed output reports the number of processed rows and the execution time per operator.

Role in the Project This component ensures that the database layer behaves as expected and that query plans can be reliably generated and compared. The outputs will later be used to align relational operators with their LLM-based equivalents within the GALOIS pipeline.

4.5 Logging & Observability

Configuration. Logging is centralized in `src/utils/logging_config.py`. It creates a `logs/` folder at runtime and registers two sinks:

- **Console** (level `INFO+`, colored): live progress while running.
- **File** `logs/pipeline.log` (level `DEBUG+`): persistent, rotated at 5 MB, retained for 10 days, compressed as `.zip`.

Python warnings are redirected to the logger so that `UserWarning`/`DeprecationWarning` also appear in the logs.

How to change verbosity. To reduce console noise, change the console sink level to `WARNING` in `logging_config.py`. The file sink will still persist `DEBUG+`:

```
logger.add(sys.stdout, ..., level="WARNING")
```

4.6 Evaluation

We have the **galois_eval.py** provided by Tutor, that is a command-line evaluator that computes performance metrics for query results across different datasets. Briefly it standardizes different result formats, normalizes textual/numeric data, computes multiple precision–recall-based metrics, aggregates them across datasets, and outputs results in several human- or machine-readable formats.

5 Results and Statistics

5.1 Statistics for the datasets in the experiments

Table 3 provides an overview of the datasets used our project. The datasets are divided into two main categories based on the type of task: IK (Incomplete Knowledge) and MC (Multiple Choice). The MC-type datasets include PREMIER, sourced from BBC, and FORTUNE, based on Kaggle data.

Dataset name	**Dataset source**	**# of queries**	**Avg. expected cells**	**Type**
FLIGHT-2	Spider [68]	3	1.0	IK
FLIGHT-4	Spider [68]	3	534.0	IK
FORTUNE	Kaggle	10	11.1	MC
GEO	Spider [68]	32	23.1	IK
MOVIES	IMDB	9	50.9	IK
PREMIER	BBC	5	57.4	MC
PRESIDENTS	Wiki	26	41.2	IK
WORLD	Spider [68]	4	33.2	IK

Tabella 3: Description of used datasets

For the **Avg_expected_cells** parameter we have implemented an ad hoc script that calculates it, more technically the **avg_cells_metric.py** calculates for each dataset the sum of the cells returned by the results of each query and then adds them together. Finally, it calculates the metric by dividing the total number of cells by the number of queries for that specific dataset. In summary this table highlights the diversity in dataset complexity and domain coverage, offering a benchmark for evaluating the performance of tabular data querying systems under different conditions, for example: Datasets like FLIGHT-2 involve simple, direct answers. In other hand the dataset FLIGHT-4 require highly complex and extensive responses.