VIETNAM NATIONAL UNIVERSITY HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

DOAN MINH QUAN

BUILDING A KNOWLEDGE GRAPH-BASED DATA PLATFORM FOR INTELLIGENT ANALYTICS

Major: Information Systems

Ha Noi - 2025

VIETNAM NATIONAL UNIVERSITY HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

DOAN MINH QUAN

BUILDING A KNOWLEDGE GRAPH-BASED DATA PLATFORM FOR INTELLIGENT ANALYTICS

Major: Information Systems

Supervisor:
Associate Professor Nguyen Ngoc Hoa

Ha Noi - 2025

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to the professors at the University of Engineering and Technology, Vietnam National University Hanoi, for their dedicated guidance and teaching throughout my studies and research at the university. I would like to extend my deep appreciation to Associate Professor Nguyen Ngoc Hoa and all the esteemed faculty members of the Information Systems Department. They have provided me with valuable guidance and direction in completing this thesis.

Despite my best efforts to complete the thesis to the best of my ability, there are still some shortcomings. I sincerely hope to receive feedback and suggestions from my professors, colleagues, and friends to improve the thesis.

Thank you very much!

Ha Noi, May 2025

Doan Minh Quan

AUTHORSHIP

I hereby declare that all the results reported in this thesis are the outcome of my own work under the guidance of Professor Nguyen Ngoc Hoa.

All references to related studies have been clearly cited and listed in the reference section of the thesis. No part of this thesis has been copied from any other work without clear acknowledgment of the source. Thesis uses AI tools like ChatGPT to correct English grammar, expression errors and advise on necessary contents. All experimental results reported in this thesis have been conducted and statistically analyzed based on the actual experimental data.

Author

Doan Minh Quan

BUILDING A KNOWLEDGE GRAPH-BASED DATA PLATFORM FOR INTELLIGENT ANALYTICS

Doan Minh Quan

Abstract In today's digital age, organizations are using and storing more and more data from various sources such as social networks, IoT devices, financial transactions. A report by IDC estimates that the Global Datasphere will reach 175 zettabytes by 2025 [15]. This data can be stored in different storage systems such as traditional databases, physical storage systems, cloud storage systems to serve business needs such as analysis, visualization, monitoring. When having to manage large amounts of data from many different sources, it is easy to lead to data loss, redundancy, or poor quality. An ideal solution to this situation is a modern data platform that integrates a metadata-based knowledge graph. A modern platform that manages all source, processing, storage, and analysis systems in a common environment allows for the connection between departmental data to find insights more easily. In addition, integrating a metadata-based knowledge graph helps users easily find the root information for the problem they are facing. This will help users easily answer questions such as: Where does this data come from, who manages it? What datasets is this dashboard created from, where is it stored? with just a simple graph query. As a result, organizations can improve governance, analysis, decision making, and cost optimization based on this approach.

Contents

List of Figures

List of abbreviations

KGs	Knowledge Graphs
AI	Artificial Intelligence
SQL	Structured Query Language
IoT	Internet of Things
ETL	Extract, Load, Transform
NoSQL	Not Only SQL
vDFKM	VNU Data Fabric-based Knowledge Management Platform
DAG	Directed Acyclic Graph
AWS	Amazon Web Services
ACID	Atomicity, Consistency, Isolation, Durability
PII	Personally Identifiable Information
API	Application Programming Interface
IA	Intelligent Analytics

INTRODUCTION

Thesis Context

In today's digital age, organizations use and store an increasing amount of data from a variety of sources, including social networks, IoT devices, and financial transactions. This data can be kept in a variety of storage systems, including traditional databases, physical storage systems, and cloud storage systems, to meet business requirements such as analysis, visualization, and monitoring.

When dealing with enormous amounts of data from multiple sources, it is easy to lose data, duplicate it, or produce poor quality results. Like other organizations, universities also face similar data problems. With diverse departments and data sources on students, curriculum, operations, etc., there is a need to manage everything on one platform. A modern data platform with a metadata-based knowledge graph is the appropriate solution to this problem. In this thesis context, we would like to implement a data solution to deal with above problems.

Research Challenges

Despite the development of data technologies, public research on solutions related to metadata knowledge graphs is quite limited. Current popular solutions also revolve around paid commercial services such as Stardog, Enterprise Knoledge. Therefore, there is a need for an open-source solution that can be easily integrated into existing data platforms, ensures security, and can be easily replaced in the future.

Thesis Objective and Main Contents

This research aims to design and implement a metadata knowledge graph-based data platform that can be easily integrated in modern platform. This research also develops Intelligent Analytics feature, which is an LLM module supporting querying and generating visulizations.

The main contents of the thesis include:

- Introducing the definition of metadata knowledge graph, intelligent analytics and benefits of integrating it into an intelligent data platform.

- Designing the system architecture and technology stacks.
- Implementing and installing the platform.
- Evaluating the platform's effectiveness with an university use case.

Thesis Structure

Apart from the introduction and conclusion, this thesis is organized into 3 main chapters with the following contents:

- Chapter 1 introduces the background and fundamental concepts related to knowledge graphs and intelligent analytics.
- Chapter 2 describes the proposed data platform architecture and its integration of metadata knowledge graph and intelligent analytics module.
- Chapter 3 presents the experimental setup, implementation, and evaluation results of the proposed system.
- The conclusion summarizes the main findings, contributions, and outlines potential directions for future research.

Chapter 1

FUNDAMENTAL THEORIES

1.1 Overview of Knowledge Graph

1.1.1 Definition

A Knowledge Graph (KG) is a network representing real-world entities and relationships between them. Based on an original entity, the Knowledge Graph provides information about every other entity in the network. One of the most popular Knowledge Graph was introduced in 2012, by Google, which is assumed that it enables people to search for things, people, locations that Google knows about with an instant speed of their query. [12]

In the process of building a data platform, managing different applications with different tasks becomes challenging because organizations could not monitor ETL processes in one tool, which can lead to problems in operation and maintenance. To solve this issue, we utilize Knowledge Graph as a visualization solution to reflect entities-data applications- and their connection in one graph.

When integrating with Intelligent Analytics or an AI module, Knowledge Graph provides meaningful relationships between services in the system. AI module through the provided contexts could help people answer questions about the platform, detect the root cause of abnormal issues immediately to make decisions.

1.1.2 Components and Structure of a Knowledge Graph

A stadard knoledgraph consists of four main components:

- **Entities**: These are real-world entities, such as people, places, concepts, objects, and events, represented as nodes in the graph.
- **Relationships**: They define how entities are connected to each other. Each edge in the graph has meaning, representing a specific relationship between two entities.
- **Labels**: They identify key attributes and properties of nodes and edges, providing additional semantics for the relationships between entities or for the entities themselves.

- **Ontology/Schema**: Ontology is an extended component of complex knowledge graphs. It includes vocabularies that represent relationships and entity types (e.g., *Person*, *Organization*, *worksAt*), and logical rules (e.g., one-way, two-way, transitive) that enhance the semantic richness of the knowledge graph. [13].

Based on the core components of a Knowledge Graph, the most efficient and widely used representation style is the RDF (Resource Description Framework) triple format, structured as: *(subject, predicate, object)*. For example, we can use the triple *(Elon Musk, follows, Sam Altman)* to represent a social relationship on X (formerly Twitter). This triple-based structure allows the graph to be scalable and queried with graph query languages like SPARQL.

1.1.3 Construction Techniques of Knowledge Graphs

Modern Construction Approaches

Since its first introduction by Google in 2012, the techniques used to build Knowledge Graphs (KGs) have improved significantly. In the earliest version, Knowledge Graphs were manually constructed by experts in various fields. Specifically, they encoded domain-specific knowledge into triples (*subject–predicate–object*) based on predefined ontologies. This method ensured high semantic detail and accuracy but was time-consuming and encountered challenges when scaling to large sizes or when the domain of knowledge was highly flexible.

To improve these weaknesses, semi-automatic approaches began to emerge, using techniques such as pattern matching, entity recognition, and database mapping to generate triples. This approach forms the basis for modern methods, leading to the construction of a smarter, more automated, and easily scalable Knowledge Graph. The metadata Knowledge Graph is built on the basis of this approach.

Metadata-based Knowledge Graph Construction

The metadata-based Knowledge Graph construction process leverages metadata generated during the operation of data systems to create connection between data services. For example:

- Orchestrators (e.g., Apache Airflow) generate metadata about the execution of tasks and the order in which data transformation tasks are run.
- Query and transformation tools (e.g., Trino, Spark) generate metadata about changes after tables are transformed, aggregated, and calculated.
- BI tools (e.g., Metabase, Looker) generate metadata about the origin of charts and dashboards, specifying which datasets they are based on and which storage they are stored in.
- Storage systems record metadata about the historical changes of schemas through transformation stages, sometimes including the format and storage type of the data.

1.2 Overview of Intelligent Analytics

Intelligent Analytics (IA) refers to the use of artificial intelligence (AI) techniques such as machine learning, natural language processing, large language models (LLMs), and semantic reasoning in data analysis. The goal is to automatically discover insights, detect patterns, create context-aware queries, and support real-time decision-making. Unlike traditional analytics that relies on predefined queries and manual interpretation, IA systems can interact with users using natural language, understand intent, and provide actionable feedback. Modern IA solutions often combine natural language interfaces with contextual metadata to generate relevant insights, enhance user interaction, and make results easier to understand.

In contrast to traditional analytics, this method is primarily descriptive and reactive, using dashboards or reports built on static data pipelines. It often requires expert users to be familiar with the schema and write precise queries. On the other hand, Intelligent Analytics emphasizes adaptability, predictive modeling, and context-aware intelligence. Supported by LLMs and enriched with metadata-based knowledge (e.g., column definitions, data lineage, semantic types), IA systems can generate semantic queries and automate analysis tasks with minimal user expertise. This reduces cognitive load on users and bridges the gap between business users and technical systems.[20]

In the current context, as organizations face an increase in the volume, speed, and variety of data, IA is becoming essential to unlocking deeper insights and supporting automated decision-making. More importantly, in collaborative human-AI scenarios, LLMs integrated with metadata-based Knowledge Graphs enable explainable analytics and dynamic user interaction. These systems not only answer queries but also suggest follow-up questions, guide users to explore data, or explain data provenance based on structured metadata. This represents a shift from static BI to a more intelligent, conversational, and adaptive analytics environment.

1.3 Role and practical application

1.3.1 The Role of Knowledge Graphs in Intelligent Analytics

A Knowledge Graph (KG), especially a metadata-based graph, plays a crucial role in powering Intelligent Analytics through structured, semantically-rich representations of data entities and their relationships. When integrated with Large Language Models (LLMs), KG allows systems to understand user intent more accurately, generate precise queries, and provide context-aware insights. For example, in natural language interfaces, KG helps differentiate terms, understand hierarchical relationships, and map user queries to the correct data sources[20].

1.3.2 Case Studies from Leading Organizations

Google's Knowledge Graph in Search and Analytics Google's Knowledge Graph strengthens the ability to provide accurate and contextually relevant results in its search engine. By understanding entities and their relationships, Google can deliver direct answers, clarify queries, and enhance the user experience. This approach has played a crucial role in the shift from keyword-based search to semantic search, enabling more natural and intuitive interactions.

Microsoft's GraphRAG for Enhanced Data Discovery Microsoft Research introduced GraphRAG, a technique that combines KGs with LLMs to improve data discovery in private datasets. By constructing a knowledge graph from textual data and leveraging it during query time, GraphRAG enhances the relevance and accuracy of responses. This method demonstrates significant improvements in answering complex queries, highlighting the synergy between KGs and LLMs in analytics [21].

Amazon's Commonsense Knowledge Graph for Recommendations Amazon has developed a knowledge graph that integrates product knowledge with commonsense reasoning to enhance its recommendation system. By encoding relationships between products and their applications, the graph enables the system to generate more informed and personalized suggestions. For example, linking "non-slip shoes" to "pregnant women" through a "used by" relationship allows for more accurate and context-aware recommendations [22].

Remarks: Integrating Knowledge Graphs into Intelligent Analytics systems enhances the capabilities of LLMs in understanding, querying, and interpreting complex data. By providing clear context, enabling explainability, and improving functionality, Knowledge Graphs serve as a core component in developing more intelligent and user-centric analytics.

1.4 Chapter Summary

This chapter has provided a solid overview of Knowledge Graphs and Intelligent Analytics, serving as a conceptual basis for the proposed data platform. We began by defining Knowledge Graphs as organized, semantic representations of real-world entities and their interrelations, emphasizing their ability to unify heterogeneous data sources and provide contextual meaning. Key components such as entities, relations, triples, and ontologies were discussed, highlighting their roles in enabling machine-readable and interoperable data. We explored different ways to build Knowledge Graphs—ranging from manual and semi-automated techniques to approaches powered by large language models—highlighting the growing importance of metadata-driven graphs in enterprise data ecosystems.

In the second half, we introduced the concept of Intelligent Analytics, which extends traditional data analytics with AI-driven capabilities such as automated insights, natural language querying, and context-aware dashboards. We examined how Knowledge Graphs serve as enablers of intelligent systems by improving query accuracy, enhancing explainabil-

ity, and connecting LLMs with rich semantic context. Through case studies from Google, Microsoft, Amazon, and Siemens, we demonstrated the practical value of integrating Knowledge Graphs into analytics workflows. This sets the stage for the following chapters, where we will explore the system architecture and specific implementation of a metadata-based Knowledge Graph for analytics tasks.

Chapter 2

KNOWLEDGE GRAPH-BASED DATA PLATFORM FOR INTELLIGENT ANALYTICS

2.1 Related Work and Base Architecture

2.1.1 vDFKM: A Data Fabric Framework

Our system architecture is referenced by a framework called vDFKM: DATA FABRIC-based DATA PLATFORM. [19] This is a standard platform based on data fabric architecture applied in enterprise and organizational environments when it is necessary to build a centralized, secure, and fast data platform for in-depth analysis purposes. Based on this framework, we have a professional platform that can easily integrate advanced features such as Knowledge Graph or Intelligent Analytics. The architecture of this framework is as follows:

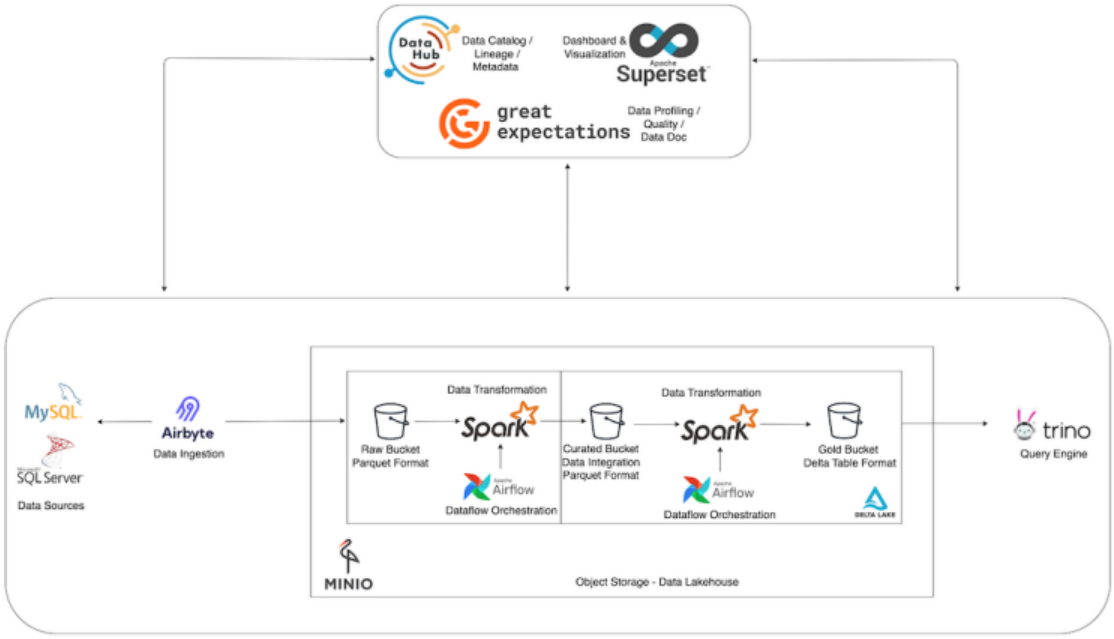

Figure 2.1: vDFKM Framework Architecture

2.1.2 Proposed High-Level Architecture

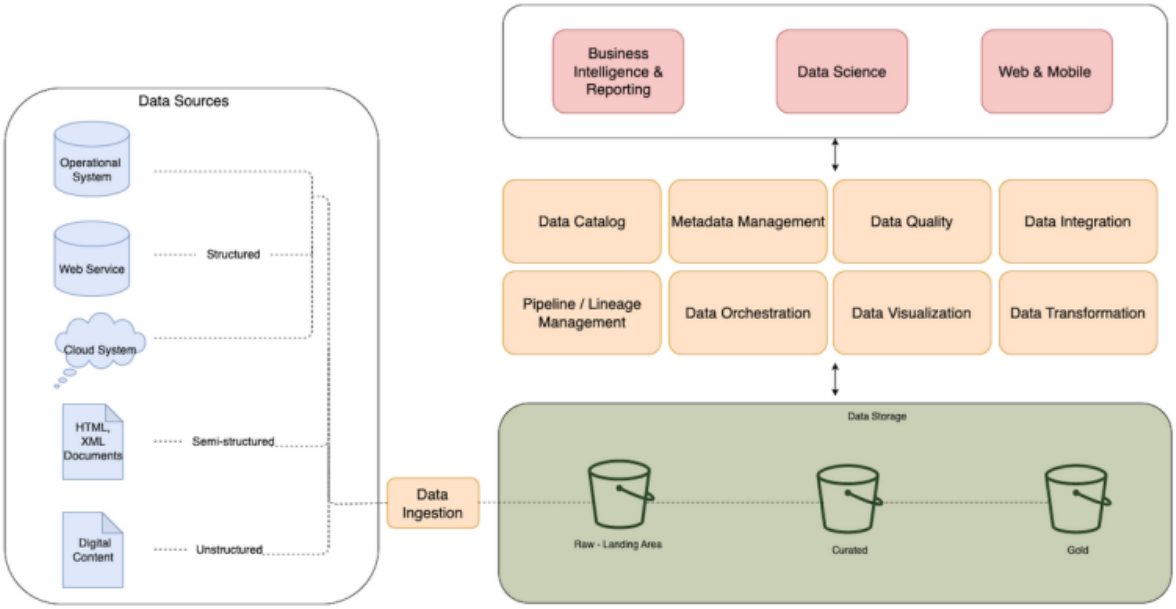

Figure 2.2: High-Level Architecture of the Framework
[19]

Based on the architecture of the reference framework, the proposed platform consists of five major layers that work together to support intelligent analytics enriched by semantic context. At the base, the *Data Ingestion Layer* leverages Debezium and Apache Kafka to

continuously capture real-time changes from source systems such as MySQL or SQL Server using Change Data Capture (CDC). These changes are streamed into the platform and passed along to the storage layer.

The *Data Storage and Lakehouse Layer* integrates MinIO as an object storage system and Apache Hudi for managing datasets with ACID guarantees and versioning, forming the backbone of a scalable and flexible lakehouse. This layer enables both batch and real-time access to raw and curated data.

Above this, the *Data Transformation Layer* utilizes Apache Spark for distributed processing, transforming raw ingested data into structured and analytics-ready formats (e.g., curated, gold datasets). Apache Airflow orchestrates the entire ETL pipeline, ensuring automated, reliable data workflows.

The semantic context and governance of the data are managed in the *Metadata and Knowledge Graph Layer*. DataHub plays a central role here by cataloging datasets, tracking lineage, and building a metadata knowledge graph that captures relationships among data assets. Great Expectations enhances this layer with data validation and profiling, ensuring that the data meets quality standards before it is consumed in analytics.

At the top, the *Analytics and Visualization Layer* enables end-users to interact with the data through query engines like Trino and visualization tools like Metabase. Integrated LLMs allow users to pose natural language questions, which are translated into executable queries with the help of contextual metadata from the knowledge graph. This integration bridges the gap between business users and complex data systems, delivering intelligent, self-service analytics and interactive dashboards.

2.2 Core Layers of the Data Platform

2.2.1 Data Source Layer

The Data Source Layer is where all the platform's data sources are centered. It can include a variety of different sources, ranging from relational databases to APIs, IoT sensors, web logs, or enterprise applications. These data sources can be structured, semi-structured, or unstructured and require enough integration flexibility to handle each type. Structured data are organized in a specific format (tables, columns), with predefined constraints, and easy to manage through traditional database management systems. Next, semi-structured data have some organization but not enough to be considered structured. Formats such as XML, JSON, and HTML are common for semi-structured data. Finally, unstructured data typically lack any rules or organization and can be in various formats. This type of data includes social media posts, image files, audio files, or unstructured text. In the context of a data platform integrated with a knowledge graph based on metadata, the Data Source Layer can fully leverage its capabilities by providing information about the types of data ingested into the system, helping businesses easily track and detect data format issues.

2.2.2 Data Ingestion Layer

The Data Ingestion Layer is the bridge between the Data Source Layer and downstream systems, which may include a data storage system or a data processing pipeline. This layer performs the process of importing, acquiring data from various sources to the destinations for further analysis. In today's complex business problems, this layer can include many tasks, ranging from extracting data with various structures from different sources to transforming it to meet business requirements. This layer also requires a design that is both scalable and flexible for potential changes in business requirements or the system architecture.

Data Ingestion tasks can be divided into two types: **batch ingestion** and **streaming ingestion**. Batch ingestion is applied when a large volume of data is collected and processed at once in batches, typically on a scheduled basis, and is intended for in-depth analysis and reporting. On the other hand, streaming ingestion refers to the immediate ingestion and processing of data as soon as new data are updated. This method is typically used in scenarios that require real-time updates and processing, such as sensor systems or anomaly detection.

Each ingestion scenario requires a different quality for the services in this layer. Batch ingestion requires services to ensure high throughput and efficient processing time when ingesting large volumes of data at once, avoiding "bottlenecks" during ingestion. In contrast, Streaming ingestion demands durability and flexibility, as it handles continuous streams of smaller data chunks and route them to the correct destination tables. Moreover, with the growing diversity of data sources, ingestion services should be able to handle various endpoints such as APIs, IoT devices, message queues, and cloud storage sources [11].

2.2.3 Data Storage and Lakehouse Layer

The Data Storage and Lakehouse Layer is the foundational component of any modern data platform. This layer is responsible for storing and managing data, including raw, semi-structured, and processed datasets. In addition, it also has responsibility for ensuring data compliance, integrity and security. This storage component must meet requirements for reliability, high availability, and cost efficiency—especially in platforms that integrate intelligent analytics and knowledge graphs.

In enterprises, storage systems are divided into two popular types: block storage, used for high-performance transactional systems, and object storage, designed for scalable, distributed data access. Object storage stores data in individual units on the same level. It does not use folders or subdirectories to organize data and stores data in multiple files. Metadata is included in each object to define file processing and usage. Object storage allows users to generate fixed-key metadata values or construct metadata by specifying both the key and value for each object. Block storage divides data into blocks and saves them with unique identifiers. When a user requests a file, the storage system combines blocks into a single unit. Block storage offers advantages such as speed, reliability, and high performance with minimal

latency for data retrieval. It is also simple to modify because no new blocks are needed to create. However, many organizations are moving away from block storage because of its lack of metadata. In modern architectures, object storage has become the preferred choice due to its ability to store large-scale unstructured data with minimal cost and excellent horizontal scalability.

Data lakes, data warehouses, and data lakehouses are three popular techniques to managing and processing huge amounts of data in businesses. A data lake is a storage repository for large amounts of raw data, including structured, semi-structured, and unstructured data. In contrast, a data warehouse is a centralized collection of data that has been optimized for querying and reporting. Data is extracted from various sources, translated into a consistent format, and loaded into the warehouse for analysis and decision-making. Business intelligence teams generally employ data warehouses to provide reports and visualizations that aid decision-making within firms. A data lakehouse is a hybrid architecture that combines the scalability and flexibility of a data lake with the data management and transactional capabilities of a data warehouse.[19] It enables users to store raw data in a data lake while also querying and analyzing the data using SQL-based tools. The data lakehouse is meant to provide a flexible and scalable platform for data processing and analysis. It can adapt to changing business needs.

In the context of knowledge graph-based intelligent analytics, this layer plays a pivotal role. It supports the storage of entity-centric datasets, relationships extracted from raw data, and snapshots of various data states that are useful for temporal reasoning. Additionally, it serves as the foundation for metadata harvesting processes that populate the knowledge graph. The structured, versioned nature of lakehouse tables ensures reproducibility and traceability, which are essential for trustworthy analytics.

2.2.4 Data Transformation Layer

The Data Transformation Layer plays a critical role in the data stream by converting raw data into clean and structured format that are suitable for analytics and downstream processing.

Transformation involves the cleaning, normalization, deduplication, and reshaping of data. Raw data from diverse sources often contains missing values, inconsistencies, or irrelevant fields. Data normalization ensures that values are represented in consistent units and formats, while schema alignment maps varying structures into unified formats, making data interoperable across systems. Additionally, timestamp standardization and key harmonization (such as aligning customer IDs across systems) are essential for coherent data fusion.

In addition, transformation also includes aggregating metrics (e.g., calculating average spending per customer) and inserting metadata (e.g., time_updated, type_updated). Additional metrics help to minimize complexity of queries, leading to speed improvement. Metadata included enable user to track the status of data imported and detect abnormal data as soon as possible. Modern transformation workflows may also incorporate real-time streaming

capabilities to enable continuous updates of data lakes or graph structures, ensuring that analytics are always performed on the most recent data.

Thus, this layer not only prepares data for efficient storage and querying but also optimizes aggregation and metrics for analytics systems. It is the foundation for building this data platform.

2.2.5 Metadata and Knowledge Graph Layer

Metadata and knowledge graph layer is one of the most important layers in the data platforms by organizing, contextualizing metadata with meaningful links. This layer integrates with all the other layers to aggregate metadata throughout the system and visualize them into a knowledge graph. This layer will recognize entities such as datasets, jobs, dashboards, etc. and the relationships between them. Advanced properties of entities such as owners or tags will also need to be carefully configured to ensure accurate information for the platform. Therefore, the choice of technology in the layer needs to satisfy integration with a variety of other tools, as well as high security to keep metadata safe.

From a practical standpoint, the Metadata and Knowledge Graph Layer supports advanced capabilities such as data discovery, impact analysis, and recommendation systems. It provides access to information about datasets, including schemas, data quality, update frequency, business definitions, ownership, and lineage. This enriched context is critical for both technical users (e.g., data engineers, analysts) and non-technical stakeholders (e.g., business users) to effectively utilize data. For example, we can query for "The dataset most used in the overview student dashboard" in the university context and the knowledge graph can return instant result instead of manually tracking in BI tools. Such scenario would be impossible with traditional relational metadata stores. This layer also contributes significantly to data governance and compliance. Policies such as access control and data retention rules can be encoded as constraints in the graph, enabling detection and auditing.

These core functions demonstrated the importance of the metadata and knowledge graph layer within the platform. Its role extends across both operational efficiency and security maintenance.

2.2.6 Analytics and Visualization Layer

This layer includes visualization tools that allow users to explore and comprehend their data visually. These technologies can help users uncover trends, patterns, and anomalies in their data and efficiently communicate insights to stakeholders. Data visualization technologies like Power Bi, Tableu, Apache Superset, and Metabase give users a set of tools for examining and analyzing data on a data platform. These technologies are crucial for supporting data-driven decision-making because they allow users to gain insights and value from their data.

An aspect in this layer supporting analytics tasks is LLM module, which helps translate

user questions into query and command to generate visualizations. This approach makes it easier for people to get started and allows more people across the company to access data. Instead of queries like "Select sum(...) from ...", user can just simply asking the module "Aggregating the student information". These approaches work best when paired with a strong semantic storage that provides aggregated metrics to optimize queries and thinking process of LLM. [7].

2.2.7 Data Flow in the Platform

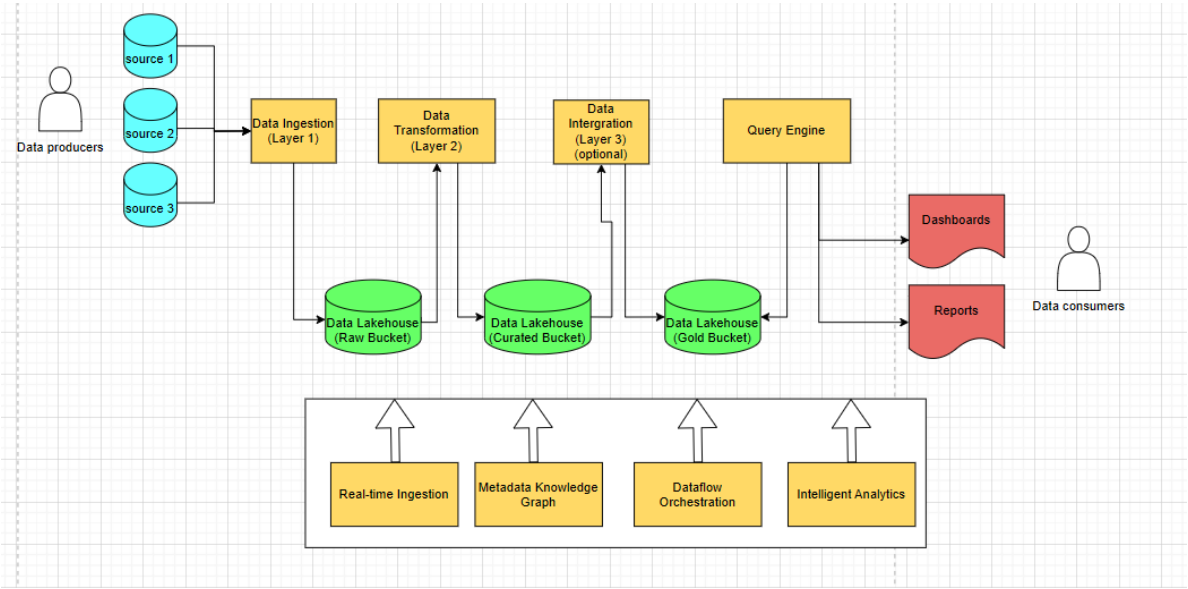

Figure 2.3: DataHub Flow in the Platform

The data flow within the knowledge graph-based data platform embodies a cohesive pipeline that transforms raw, heterogeneous data into structured, semantically rich insights. It begins at the **Data Source Layer**, where data is ingested from multiple origins, such as transactional databases, APIs, and sensor systems. The data then traverses the **Data Ingestion Layer**, which standardizes, batches, or streams the inputs while ensuring security and compliance.

Upon ingestion, the data is persisted and organized in the **Data Storage and Lakehouse Layer**, where structured, semi-structured, and unstructured formats are retained in a unified architecture supporting both analytical and transactional workloads . From here, the **Data Transformation Layer** applies cleansing, normalization, and enrichment logic—laying the foundation for semantic relationships and domain-specific logic.

Finally, insights propagate to the **Analytics and Intelligent Services Layer**, where machine learning, reasoning engines, or business dashboards transform the knowledge graph into actionable decisions [14].

This multi-stage data flow fosters end-to-end traceability, data quality enforcement, and semantic interoperability across the platform. The diagram below illustrates the flow of data and the interactions among core architectural layers.

2.3 Technology Architecture

Based on the architecture from the vDFKM framework, we changed some technologies to integrate Knowledge Graph and Intelligent Analytics into the platform while still keeping the core values of the Data Fabric architecture.

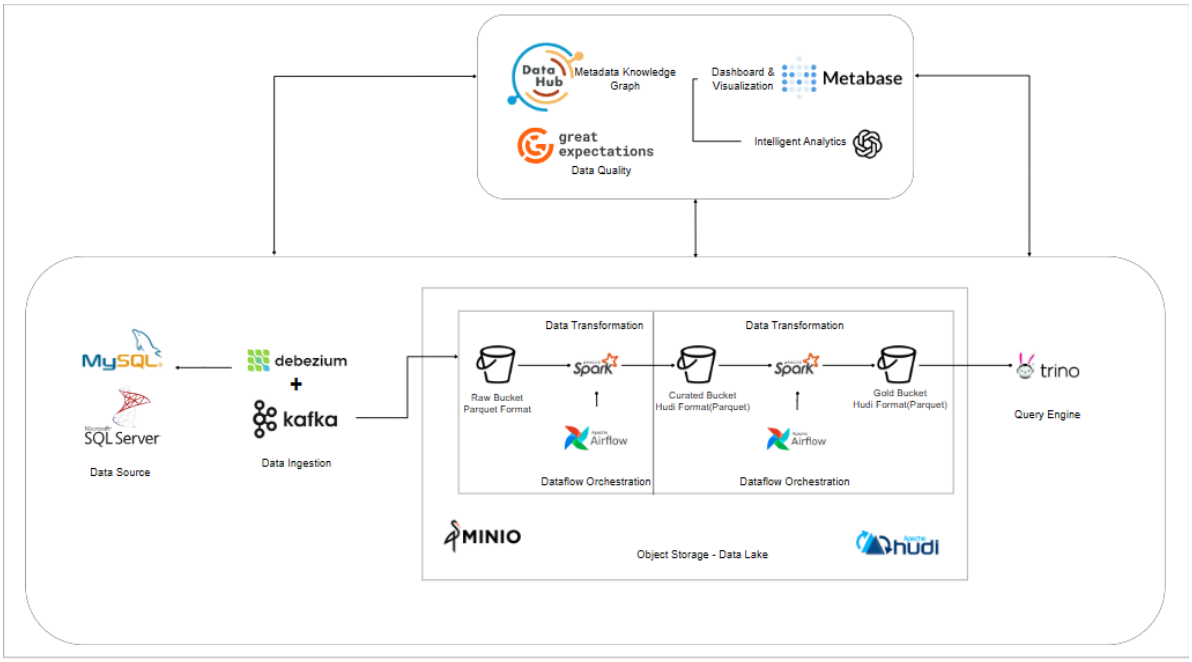

Figure 2.4: Technology Architecture

2.3.1 Data Source: MySQL and SQL Server

MySQL

MySQL is an open-source database management system developed and maintained by Oracle. It is widely used due to its ease of deployment, speed, and compatibility across various systems. MySQL supports a wide range of storage engines and replication techniques, making it suitable for distributed environments and data-driven applications. In the context of this platform, MySQL is considered a primary data source representing both batch and streaming data [18].

SQL Server

Microsoft SQL Server is a relational database engine designed to retrieve, store, and manage data for structured querying, transactional processing and business intelligence features. SQL Server integrates seamlessly with enterprise tools such as Power BI and Azure Synapse, allowing it to serve as a robust enterprise data source. Within this platform, it is used to simulate domain-specific datasets with versioned records and structured schemas [16].

2.3.2 Data Ingestion: Debezium and Apache Kafka

Debezium

Debezium is an open-source distributed platform designed for change data capture (CDC), enabling applications to react to changes occurring in databases. It taps directly into transaction logs—such as the MySQL binlog, PostgreSQL WAL, or SQL Server transaction log—to detect and stream these changes. Built on top of Apache Kafka and Kafka Connect, Debezium transforms change events into structured messages that can be consumed by downstream consumers [9].

The architecture of Debezium consists of the following core components:

- **Debezium Connectors:** Designed for specific databases (e.g., MySQL, PostgreSQL), they read changes from database logs.
- **Kafka Connect:** A framework to run Debezium connectors in a scalable and fault-tolerant manner.
- **Apache Kafka:** Acts as a central messaging hub where change events are routed to topics.
- **Schema Registry (optional):** Maintains versioned schemas (Avro/JSON) for consistent serialization and deserialization.
- **Consumers:** Downstream services that subscribe to Kafka topics to consume change events.

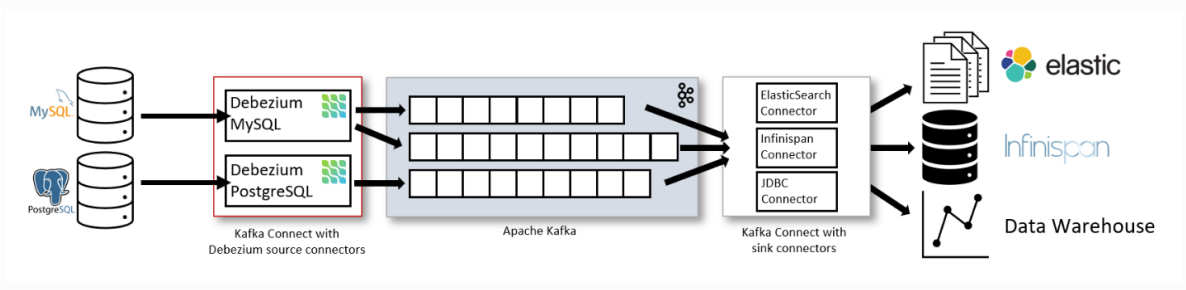

Figure 2.5: Debezium Architecture deployed with Kafka

Another way to deploy Debezium is by using Debezium Server. Debezium Server is a ready-to-use and configurable application that streams change events from a source database to various messaging services.

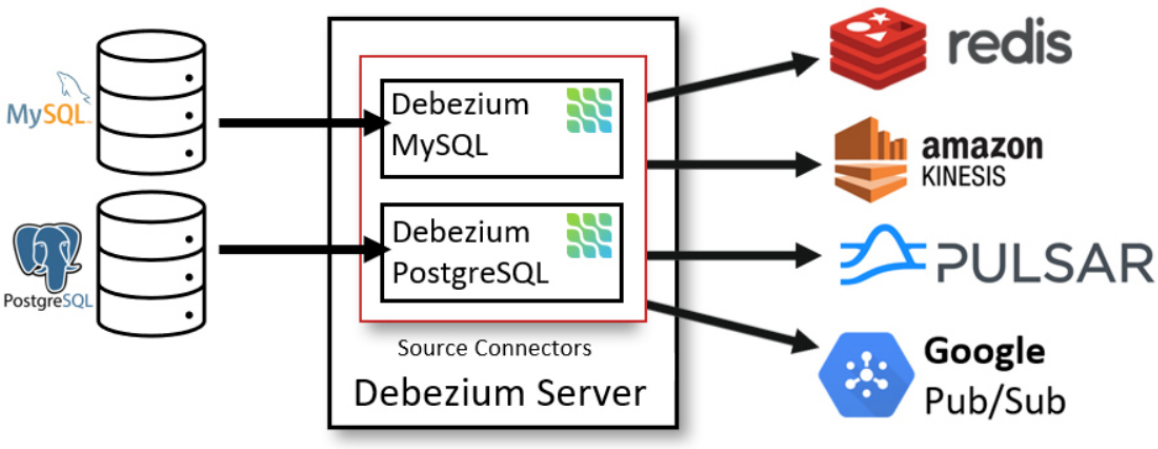

Figure 2.6: Debezium Server Architecture

The key strengths of Debezium include its low-latency log-based change data capture, along with support for schema updates. It also offers fault tolerance through offset tracking, ensuring that the system can recover from disruption without data loss or duplication. Furthermore, Debezium supports advanced CDC patterns such as transactional outbox, event routing, and change events enrichment.

Apache Kafka

Apache Kafka is a distributed event streaming platform that serves as the backbone in real-time data pipelines and streaming applications. Essentially a distributed publish-subscribe messaging system, Kafka is increasingly adopted in real-time analytics applications and event-driven architectures due to its ability to handle large volumes of data with high throughput [4]. Kafka architecture includes:

- **Producers:** Clients that publish data (events, messages) to Kafka topics.
- **Topics:** Logical channels to which records are written and from which consumers read.
- **Partitions:** Kafka topics are split into partitions for parallelism and scalability.
- **Brokers:** Kafka servers that handle incoming data and client requests; each broker hosts one or more partitions.
- **ZooKeeper:** Used for managing Kafka cluster metadata and coordination.
- **Consumers:** Applications that subscribe to topics and process messages.
- **Consumer Groups:** Enable horizontal scaling of consumers across partitions while ensuring each message is consumed exactly once per group.

In the context of this thesis, using the Kafka Connect framework, Kafka topics can receive CDC messages from Debezium connectors whenever there is an update in the source data.

These events are then consumed and appropriately transformed before being sent to the subsequent layers of the platform. Overall, Kafka is a powerful tool for the Data Ingestion Layer, capable of handling both batch and streaming processing. Kafka strongly complements the Knowledge Graph by enabling near real-time updates of CDC events onto the graph.

2.3.3 Data Storage and Lakehouse: MinIO and Apache Hudi

MinIO: High-Performance Object Storage for the Data Lakehouse

MinIO is an open-source object storage system ideal for data storage applications such as data lakes and lakehouses. As a data lake storage solution, MinIO offers a highly scalable and adaptable platform for storing massive amounts of data in variety of forms and structures. MinIO's distributed architecture enables it to extend horizontally, adding storage space and throughput as required. This makes it perfect for data lake applications where data quantities might increase quickly and unpredictably over time. MinIO can also be used as the storage layer for an integrated data platform in a data lakehouse scenario, giving a central repository for all structured and unstructured data.[17]

In terms of architecture, MinIO is typically deployed in a distributed mode for production-grade workloads. The core components of a distributed MinIO deployment include:

- **MinIO Server Nodes**: Handle object read/write operations and maintain metadata.
- **Erasure sets**: Ensures data resiliency and fault tolerance across nodes.
- **Object API (S3-Compatible)**: Provides programmatic access for applications and tools.
- **IAM and Policies**: Fine-grained access control through user and group-based policies.
- **Replication and Versioning Modules**: Enable disaster recovery and change tracking.

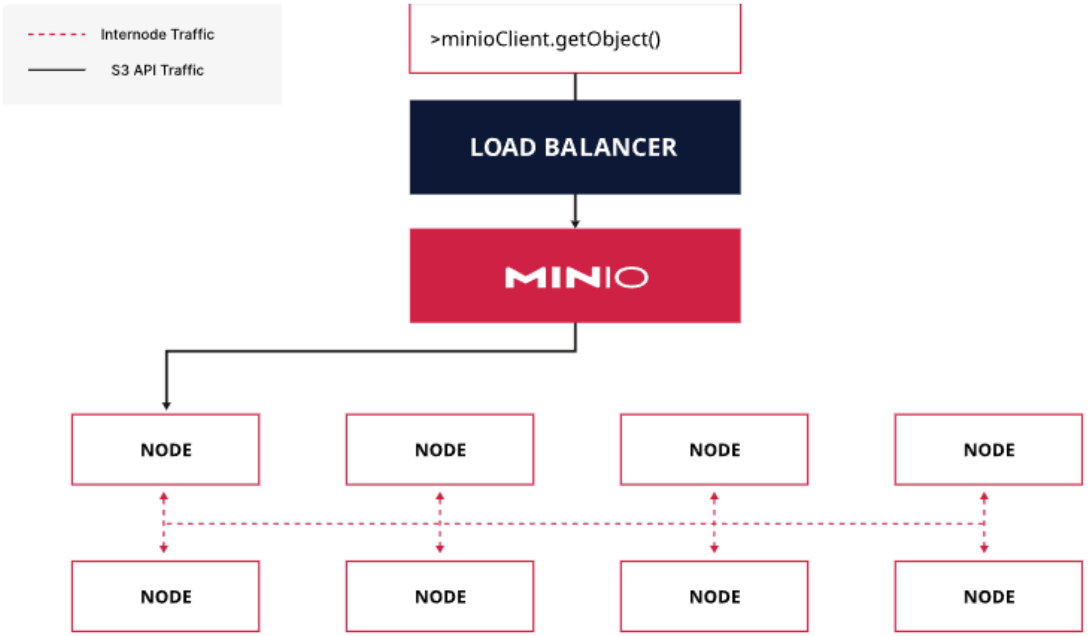

Figure 2.7: MinIO Server Nodes

The load balancer routes the request to any node in the deployment. The receiving node handles any internode requests thereafter.

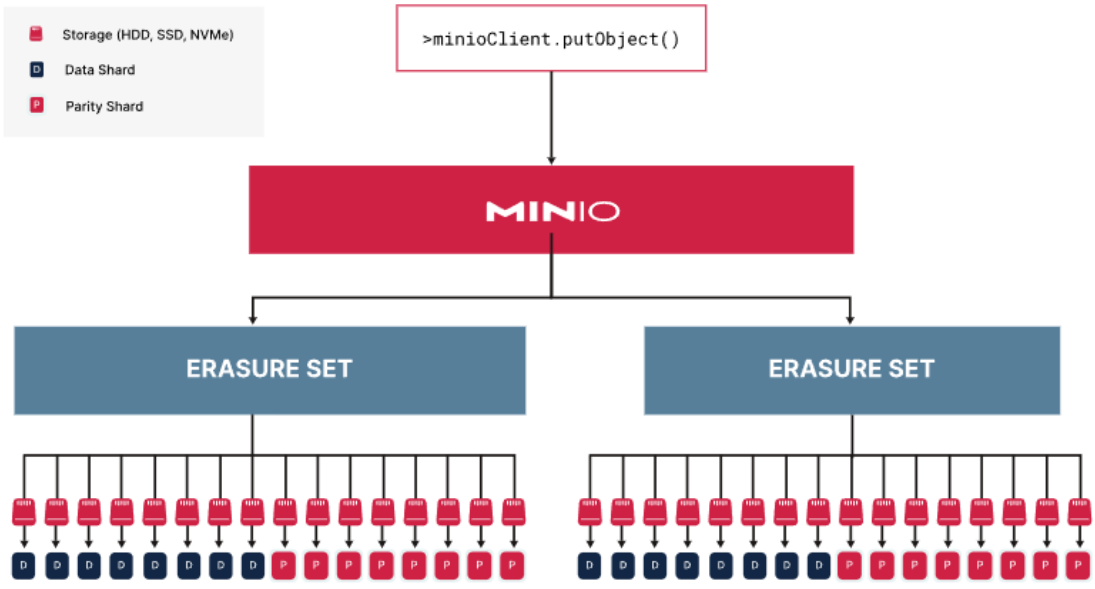

Figure 2.8: MinIO Erasure Sets

In the context of our Knowledge Graph-Based Data Platform, MinIO serves as the primary storage backend for both raw and intermediate datasets. It provides scalable and durable storage for structured, semi-structured, and unstructured data ingested from multiple sources. Additionally, its compatibility with Apache Arrow, Apache Spark, and other

tools ensures seamless integration across the analytics stack.

Apache Hudi: Transactional Data Lake Framework

Apache Hudi (Hadoop Upserts Deletes and Incrementals) is an open-source transactional data lake framework that brings database-like functionalities to large-scale data lakes. Originally developed by Uber, Hudi addresses several limitations of traditional data lakes by introducing capabilities such as record-level insert/update/delete (upsert), change data capture (CDC), and time travel queries.

At its core, Hudi enables incremental data processing and efficient data ingestion by allowing data pipelines to write changes directly into datasets stored in columnar formats like Parquet or ORC. This makes it particularly effective in scenarios involving frequent data updates, such as real-time analytics and machine learning feature stores.

Hudi supports two primary table types:

- **Copy-on-Write (COW)**: Updates create new versions of files, offering query efficiency at the cost of higher write latency.
- **Merge-on-Read (MOR)**: Updates are stored as delta logs and compacted later, providing faster ingestion with some query trade-offs.

A typical Apache Hudi architecture includes the following components:

- **Hudi Writer**: Responsible for ingesting data into the Hudi dataset (via Spark, Flink, Hive, or Presto).
- **Timeline Service**: Maintains a timeline of actions (commits, rollbacks) and enables snapshot isolation.
- **Indexing Layer**: Accelerates record location for upserts using Bloom filters, global or partitioned indices.
- **Storage Engine**: Stores data in either COW or MOR formats over distributed storage systems (HDFS, S3, MinIO, etc.).
- **Query Engines**: Spark, Presto, Hive, and Trino support querying Hudi tables with snapshot or incremental semantics.

Apache Hudi also integrates seamlessly with modern data tools such as Apache Flink, Apache Spark, and Trino, allowing scalable batch and streaming ingestion as well as flexible querying options [10].

In the context of our Knowledge Graph-Based Data Platform, Apache Hudi plays a central role in ensuring that ingested and transformed datasets remain consistent, queryable, and updatable over time. It supports real-time analytical scenarios, allows capturing evolving knowledge entities, and helps maintain data lineage through commit timelines and metadata storage. Combined with MinIO, Hudi forms the foundation of a robust Lakehouse model that unites scalable storage and structured transactional processing [23].

2.3.4 Data Transformation and Orchestration

Apache Spark: Unified Analytics Engine for Large-Scale Data Processing

Apache Spark is a popular data processing engine, ideal for large-scale data transformations. Spark provides a fast and versatile platform for parallel data processing over distributed clusters, allowing users to convert, clean, and modify data on a large scale. One of Spark's primary advantages is its ability to manage complicated data kinds and structures. Spark provides a variety of high-level APIs for working with data, including SparkSQL and Spark DataFrames.

The architecture of Spark is structured around several key components:

- **Driver Program**: The central coordinator that translates user code into tasks and schedules them across the cluster.
- **Cluster Manager**: Allocates resources across multiple Spark applications (e.g., Standalone, YARN, Mesos, Kubernetes).
- **Executors**: Distributed agents responsible for executing tasks and storing data on worker nodes.
- **Tasks**: The smallest units of work sent from the driver to executors for processing.
- **Resilient Distributed Dataset (RDD)**: The fundamental abstraction in Spark, representing an immutable, distributed collection of objects that can be processed in parallel .

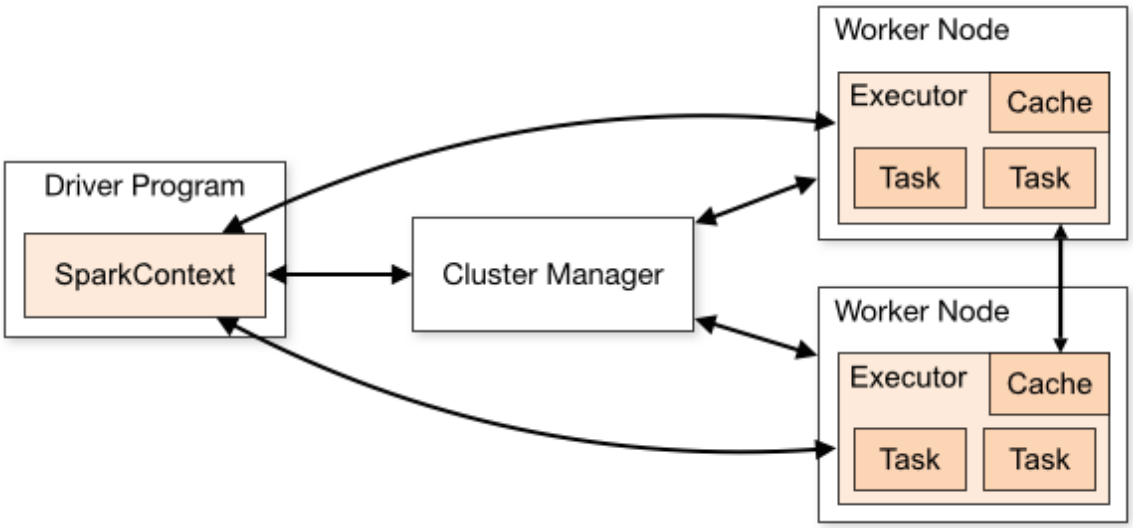

Figure 2.9: Cluster Mode Overview

Spark supports several specialized libraries that extend its capabilities, including:

- **Spark SQL**: Enables querying structured data using SQL syntax and integrates seamlessly with the DataFrame and Dataset APIs.
- **MLlib**: A scalable machine learning library providing algorithms for classification, regression, clustering, and recommendation.
- **GraphX**: Supports graph-parallel computations.
- **Structured Streaming**: Provides fast, fault-tolerant, end-to-end exactly-once stream processing [6].

In the context of this platform, Apache Spark is an effective tool for data transformation applications, offering a versatile and scalable platform for processing and manipulating data at scale. Organizations that use Spark's capabilities can increase the efficiency and accuracy of their data transformation operations, as well as generate more insights and value from their data.

Apache Airflow: Workflow Management Platform

Apache Airflow is an open-source platform for developing, scheduling, and monitoring batch workflows. It provides a Python-based framework where complex processes are expressed as Directed Acyclic Graphs (DAGs) of tasks. Each DAG file (a Python script) defines tasks and their dependencies, enabling flexible, code-driven workflows. Airflow includes a modern web UI for visualizing DAG state and logs, simplifying monitoring and debugging [5].

- **Metadata Database:** A SQL database (e.g., PostgreSQL or MySQL) that stores metadata about DAG runs, task states, and scheduling information.
- **Scheduler:** A core service that triggers DAG runs and determines which tasks to execute according to schedule or dependencies.
- **Web Server:** Serves the Airflow web interface, allowing users to inspect, trigger, and manage workflows interactively.
- **Executor:** The execution engine (e.g., Local, Celery, or Kubernetes) configured in the scheduler; it launches task instances and dispatches them to workers.
- **Workers:** Processes or containers that perform the actual task execution as directed by the scheduler/executor, enabling parallel and distributed processing.

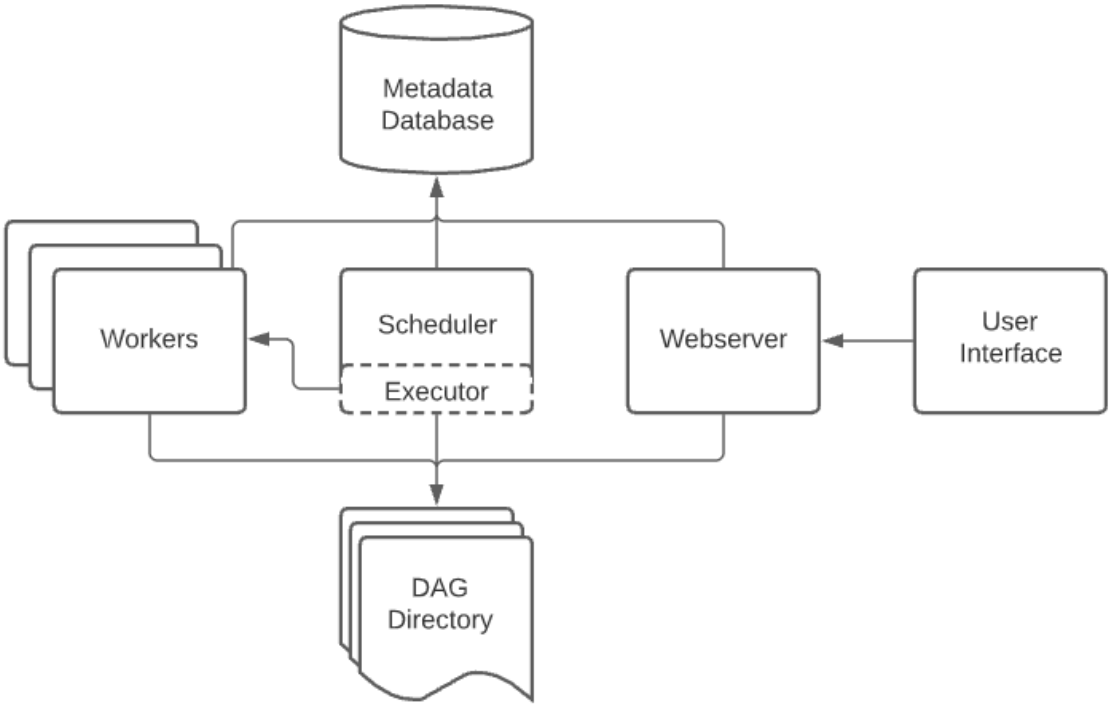

Figure 2.10: Apache Airflow Architecture Overview

Airflow's features include flexible scheduling, monitoring, and extensibility. DAGs allow specifying cron-like schedules, execution intervals, or event-based triggers, and tasks run only when their dependencies are satisfied. The web interface and CLI provide detailed logs and status views for each task, aiding observability and troubleshooting. The system is highly extensible: it offers many built-in operators (for databases, clouds, Hadoop/Spark, etc.) and sensors, and it allows custom operators and plugins for new systems.

2.3.5 Meta-Data and Knowledge Graph Management

DataHub: A Metadata Platform

DataHub is a modern data catalog designed to streamline metadata management, data discovery, and data governance. It enables efficient exploration of organizational data, lineage tracking, and enforcement of data contracts. Notably, DataHub models metadata as a graph of interconnected entities (datasets, dashboards, users, etc.) and relationships (schema, ownership, lineage).

DataHub's architecture follows a Generalized Metadata Architecture that combines relational storage, search indexing, and graph capabilities [8]. Key components include:

- **Metadata Store (GMS)**: A Java-based service exposing REST APIs to manage metadata. Entities and their associated Aspects are stored in MySQL and indexed in Elasticsearch, while all metadata changes are published to Kafka as an event stream.
- **Metadata Models**: A schema-first approach defines the metadata graph's structure.

Each Entity type (e.g., `Dataset`, `Table`, `Pipeline`) has a unique URN and can have multiple Aspects (such as schema, ownership, tags).

- **Ingestion Framework**: A modular Python library for metadata ingestion. DataHub provides connectors to extract metadata from external systems (databases, data warehouses, streaming platforms, BI tools) and write it into the metadata store via Kafka or APIs.
- **Graph Service**: An (optional) Neo4j graph database materializes the current metadata state. This allows efficient graph traversals (such as lineage queries).
- **GraphQL API and UI**: A strongly-typed GraphQL interface exposes the metadata graph for querying and mutation. The React-based UI builds on this API to provide unified search, discovery, and governance workflows.

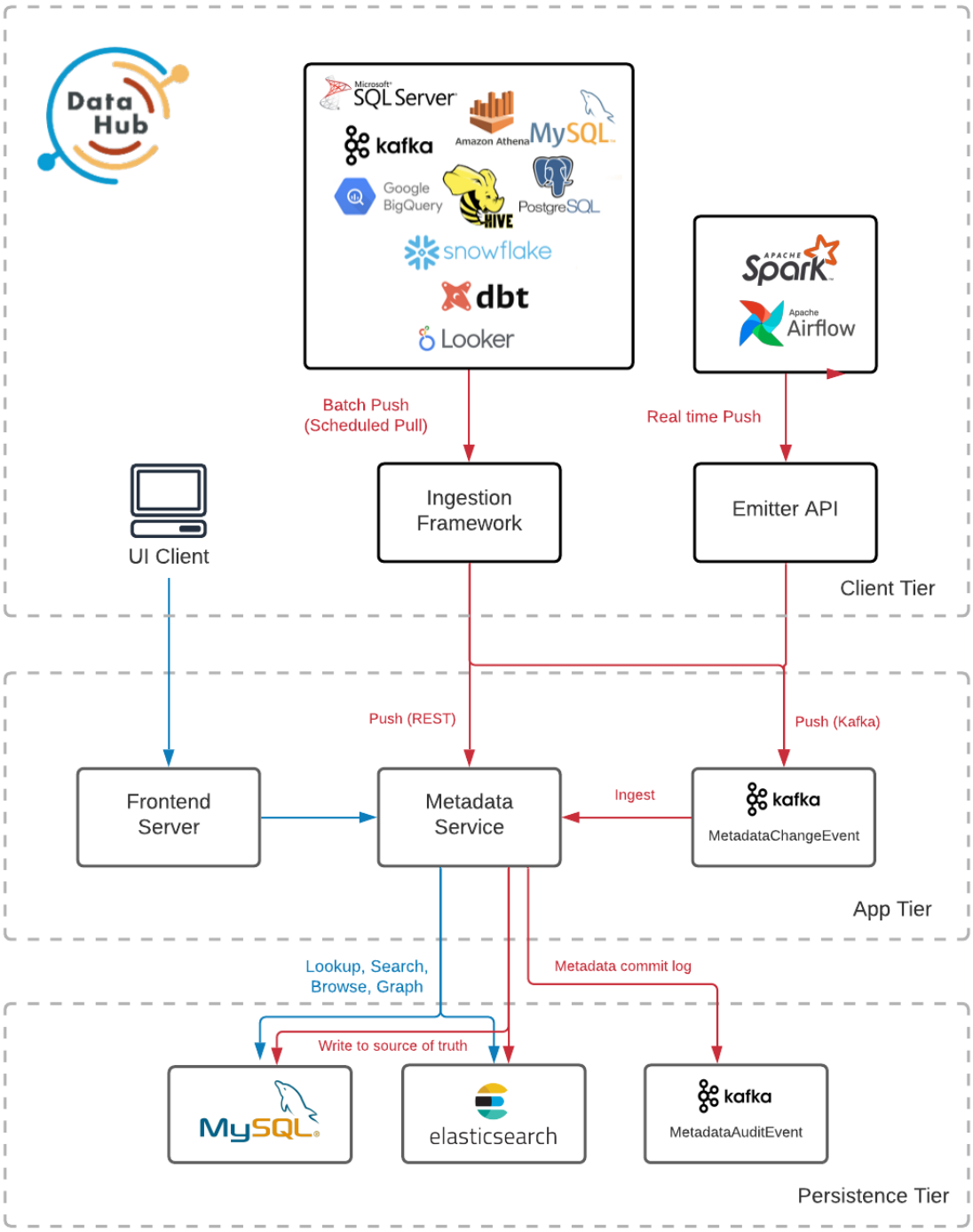

Figure 2.11: DataHub Architecture

Knowledge-Graph Integration DataHub's metadata store inherently represents a knowledge graph of the data domain. Entities and relationships form a graph structure exposed via the GraphQL API. In addition:

- Kafka-based change events allow real-time integration of metadata updates into downstream systems.
- Elasticsearch supports functions related to data search and analysis, which utilize meta-

data change in knowledge graph to gain insights.

- Federated metadata services allow teams to operate domain-specific instances, while a global DataHub instance aggregates a unified graph

Great Expectations: Data Validation Framework

Great Expectations is an open-source data validation, documentation, and profiling tool designed to help teams maintain data quality throughout the data pipeline. It offers a declarative framework where users define "expectations"—assertions about data characteristics—that are automatically checked during data ingestion, transformation, and serving.

At its core, Great Expectations treats data quality as a first-class concern by codifying tests as machine-readable contracts. These expectations are human-readable and can be version-controlled, promoting transparency and collaboration between data engineers, analysts, and business users.

Architecture The key components of Great Expectations include:

- **Data Context**: A configuration layer that manages project settings, expectations, data sources, and stores for results and validations.
- **Expectations Suite**: A collection of rules (e.g., "no nulls in primary key", "values in a column must be between X and Y") defined for specific datasets.
- **Checkpoint**: An orchestration mechanism that bundles one or more validation runs, typically triggered manually or by scheduled jobs.
- **Data Docs**: Automatically generated, interactive HTML documentation of validation results.

2.3.6 Querying and Analytics

Trino: Distributed SQL Query Engine

Trino, formerly known as PrestoSQL, is an open-source distributed SQL query engine designed for running fast, interactive analytic queries across heterogeneous data sources. It allows querying data directly where it lives—whether on object storage systems like S3, distributed file systems like HDFS, or in traditional relational and NoSQL databases—without requiring prior data movement or duplication. Trino has become a cornerstone technology in modern data architectures, particularly for organizations adopting a data lakehouse or federated query model.

At its core, Trino is optimized for low-latency and high-concurrency workloads over massive datasets. By separating storage from compute, and through a lightweight execution engine built for large-scale parallelism, Trino ensures that queries can run interactively even

at petabyte scale [3]. It supports standard ANSI SQL, including advanced features like window functions, subqueries, joins across data sources, and complex aggregations, making it familiar and powerful for data analysts and engineers alike.

The Trino architecture follows a distributed design based on a coordinator-worker model, comprising the following key components:

- **Coordinator**: The single coordinator node is responsible for parsing SQL statements, optimizing query plans, scheduling tasks, and managing metadata. It serves as the central control plane for the cluster.
- **Workers**: Worker nodes execute parts of the query in parallel, processing data locally and passing intermediate results through pipelines back to the coordinator.
- **Catalogs**: Catalogs provide mappings to external data sources, enabling Trino to organize schemas and tables through connectors.
- **Connectors**: Pluggable components that allow Trino to communicate with various systems such as Hive, MySQL, PostgreSQL, Cassandra, Kafka, MongoDB.
- **Query Execution Engine**: Handles distributed SQL processing, including pipelining, operator parallelism, and task fault tolerance.

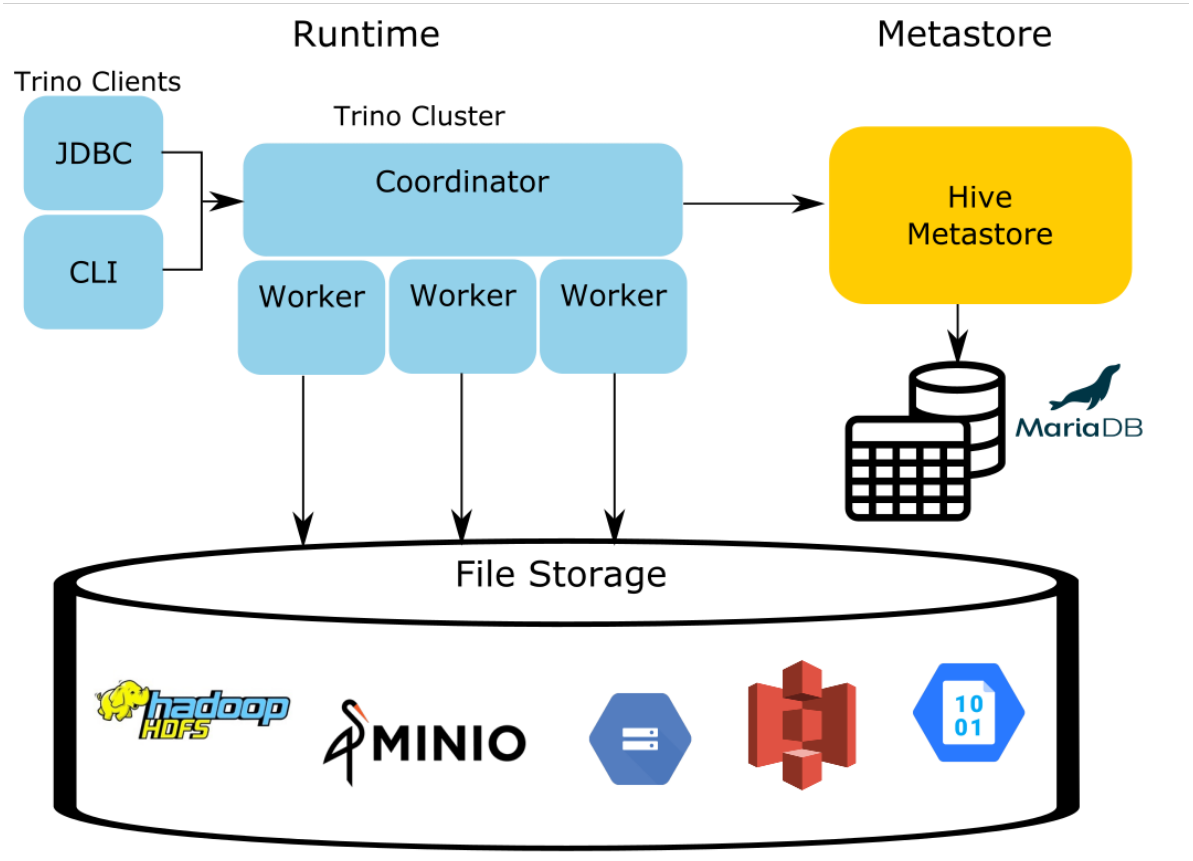

Figure 2.12: Trino Architecture

In the context of the proposed platform, Trino serves as the primary distributed query engine. It enables users to perform fast, interactive analytics across heterogeneous data sources, including the MinIO-based data lake, Apache Hudi datasets, and operational databases like MySQL and SQL Server. Its federated querying capability reduces the need for complex ETL pipelines and empowers agile data exploration and reporting directly across the entire platform.

Metabase: Open Source Business Intelligence Platform

Metabase is an open-source business intelligence (BI) and data visualization platform designed to make data exploration accessible to non-technical users through an intuitive, low-code/no-code interface. It empowers decision-making across organizations by abstracting SQL complexities and offering interactive dashboards. By supporting connections to a wide range of databases and data warehouses — including MySQL, PostgreSQL, MongoDB, Trino, and more — Metabase allows users to start querying data with minimal setup [1]. For more technical users, Metabase provides a SQL editor featuring query snippets, syntax highlighting, and parameterized dashboards.

The architecture of Metabase follows a modular and extensible model that separates concerns across several core components:

- **Application Server**: Manages the web interface, authentication, scheduling, and user sessions. It can run as a standalone service or be containerized via Docker.
- **Metadata Database**: Stores application metadata such as users, saved questions, dashboards, data models, and audit logs. It commonly uses relational databases like Postgres or MySQL in production environments.
- **Query Processor**: Handles communication with external data sources, translating user actions into optimized SQL queries for execution.
- **Scheduler and Alerts Engine**: Enables users to create scheduled reports, alerts, and subscriptions triggered by specified thresholds.
- **Visualization and Dashboard Layer**: Renders query results into various visual formats such as tables, bar charts, line graphs, scatter plots, maps, and custom visualizations.

Within the context of the proposed platform, Metabase serves as the primary interface for querying and visualizing distributed data stored across multiple systems. It connects directly to Trino to access datasets stored on MinIO, Apache Hudi, and relational databases, presenting results through dynamic dashboards. By enabling both technical and non-technical users to create, share, and automate insights without deep technical expertise, Metabase plays a critical role in democratizing data access and supporting data-driven decision-making across the organization.

2.3.7 LLM-Powered Intelligent Analytics

The emergence of Large Language Models (LLMs) has fundamentally transformed the way users interact with data systems, introducing natural language interfaces that enable intuitive and dynamic data exploration [2]. In the context of intelligent analytics platforms, LLMs such as ChatGPT play a pivotal role by interpreting user prompts, automatically generating SQL queries, recommending visualizations, and providing narrative summaries of analytical results. By abstracting the complexity of database schemas, query syntax, and visualization logic, LLMs democratize access to analytics for users across varying levels of technical expertise [24].

The architecture for LLM-powered intelligent analytics integrates LLMs into the data platform via a specialized interaction layer:

- **Large Language Models (ChatGPT)**: Serve as the core natural language understanding and generation engine, parsing user prompts into structured analytical tasks such as SQL query generation, dashboard creation, or insights summarization.
- **Integration Layer**: Acts as a bridge between the LLM and backend services, providing schema context from DataHub, ensuring query validity, and invoking visualization services in Metabase to render results.

Within the proposed platform, the LLM-powered analytics module acts as the intelligent front-end that unifies natural language querying, semantic metadata, and visualization workflows. It bridges the gap between human intent and data-driven decision-making, making analytics more accessible, conversational, and efficient across the organization.

2.4 Chapter Summary

This chapter introduced the core layers and technology stack underpinning the Knowledge Graph-based Data Platform for Intelligent Analytics. We explored each component's role — from data storage and transformation, to metadata management, intelligent querying, and LLM-powered analytics. Together, these technologies form an integrated ecosystem that enables efficient, scalable, and intelligent data workflows, providing a strong foundation for building advanced analytics capabilities within modern data-driven organizations.

Chapter 3

EXPERIMENTS AND EVALUATION

3.1 Experimental Environment

All experiments were conducted on a lightweight server equipped with an Intel Xeon (Cascade Lake) processor, 16GB of RAM, and an additional 8GB of swap memory. The system was built using Docker, virtual environments, and Python version 3.12. Here is the specific set of tools we installed in our data platform:

- MySQL Server: 0.3.0 (for Linux)
- Apache Kafka: 3.0
- Debezium: 3.0
- Minio: RELEASE.2025-04-22T22-12-26Z
- Apache Hudi: 0.15.0
- Apache Spark: 3.5.5
- Apache Airflow 2.10.4
- Trino: 400
- Metabase: 54.5

3.2 Service Integration

3.2.1 Debezium Integration with MySQL and Kafka

To initiate real-time change data capture (CDC) from the MySQL database, Debezium is integrated with both MySQL and Apache Kafka through a REST API call to the Kafka

Connect interface. The integration process begins with a POST request using the `curl` command, which registers a MySQL source connector with the Kafka Connect cluster. Below is the command:

```
$ curl -i -X POST -H "Accept:application/json" -H "Content-Type:
    application/json" localhost:8083/connectors/ -d '{
  "name": "school-connector",
  "config": {
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "tasks.max": "1",
    "database.hostname": "mysql",
    "database.port": "3306",
    "database.user": "root",
    "database.password": "123456",
    "database.server.id": "184054",
    "topic.prefix": "dbserver1",
    "database.include.list": "school",
    "schema.history.internal.kafka.bootstrap.servers": "kafka:9092",
    "schema.history.internal.kafka.topic": "schema-changes.school"
  }
}'
```

This configuration sets up a Debezium MySQL connector named `school-connector`. The connector captures all changes from the `school` database and publishes them as events to Apache Kafka topics prefixed with `dbserver1`. Each topic corresponds to a table within the database, making it possible to capture insert, update, and delete operations in real time.

The critical parameters in this configuration include:

- **connector.class**: Specifies the connector type as Debezium's MySQL connector.
- **database.hostname**, **port**, **user**, **password**: Define the MySQL instance details for establishing a connection.
- **database.include.list**: Restricts CDC to only the `school` database.
- **schema.history.internal.kafka.bootstrap.servers**: Directs the connector to use Kafka for storing schema history.
- **schema.history.internal.kafka.topic**: The Kafka topic used to persist schema changes.

After registration, the connector looks over the MySQL binary log to capture and stream data changes into Kafka topics, which are then consumed, stored in MinIO and processed by Spark in the following stages. This setup ensures low-latency, scalable, and fault-tolerant ingestion of transactional data into the data platform.

After Debezium configuration, Python scripts are run to set up Kafka consumers that subscribe to change event topics for each MySQL table. These scripts parse JSON-formatted CDC messages and write them into the raw data layer ("raw bucket") with the appropriate

schema applied. This ensures structured ingestion of raw data, secures change history, and sets the foundation for later validation and transformation processes.

Listing 3.1: Sample Kafka Consumer Script for Raw Bucket Ingestion

```
import json
from confluent_kafka import Consumer
from raw_bucket.write_to_raw_bucket import write_raw_bucket
from pyspark.sql.types import *

DATABASE_NAME = 'school'
KAFKA_TOPIC = 'dbserver1.school.<table_name>'
TABLE_NAME = '<table_name>'

SCHEMA_DEFINITION = StructType([
    # Define table-specific schema here
])

full_schema = StructType([
    StructField("schema", MapType(StringType(), StringType()), True),
    StructField("payload", SCHEMA_DEFINITION, True)
])

conf = {
    'bootstrap.servers': 'localhost:9098',
    'group.id': f'{TABLE_NAME}_consumer_group',
    'auto.offset.reset': 'earliest'
}

consumer = Consumer(conf)
consumer.subscribe([KAFKA_TOPIC])

try:
    while True:
        msg = consumer.poll(1.0)
        if msg is None: continue
        if msg.error(): continue

        try:
            if msg.value():
                message_value = msg.value().decode('utf-8', errors='
                    ignore')
                json_data = json.loads(message_value)
                write_raw_bucket(
                    data=json_data,
                    bucket="raw_bucket",
                    status="unprocessed",
                    database=DATABASE_NAME,
                    table=TABLE_NAME,
```

```
                    schema=full_schema
                )
        except json.JSONDecodeError:
            pass
except KeyboardInterrupt:
    pass
finally:
    consumer.close()
```

3.2.2 MinIO Integration

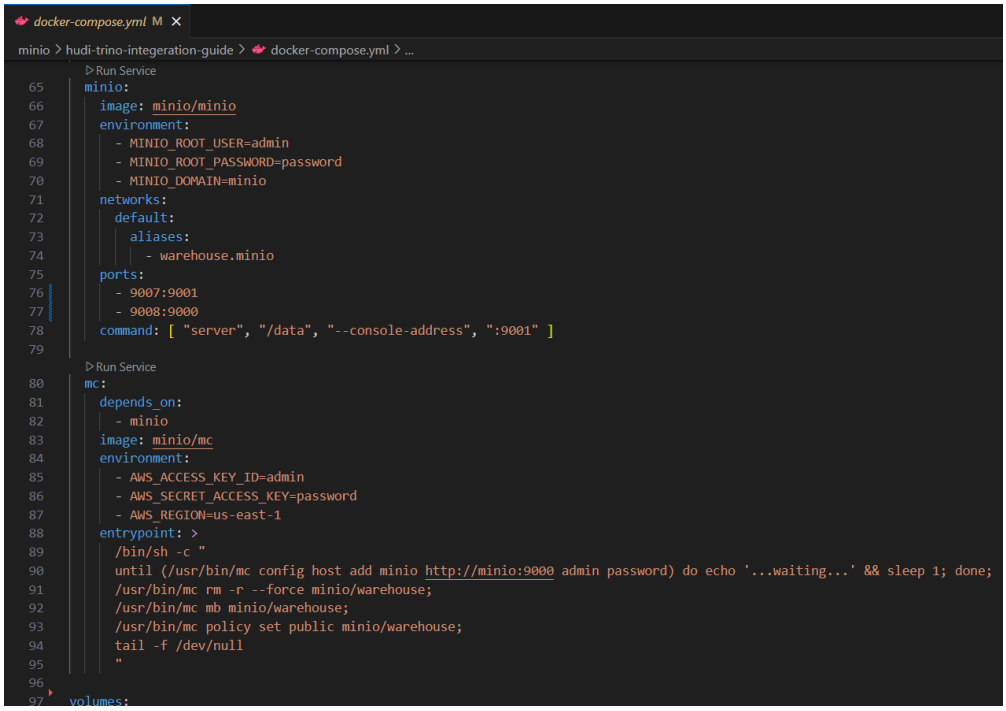

Figure 3.1: MinIO Docker Container

After messages are consumed from Kafka and parsed into JSON structured format, they are written into the raw data layer under the path `raw_bucket/unprocessed`.This folder resides in an object storage system powered by MinIO, which acts as the S3-compatible data lake. Each record is stored in its original unprocessed form, maintaining data accuracy and supporting schema evolution.

This raw zones serves as a staging area for transformation jobs. By keeping the unmodified event payloads, the platform enables auditability, reprocessing, and rollback capabilities in case of schema or logic changes. The clear separation between raw and processed layers is a key principle in the lakehouse architecture.

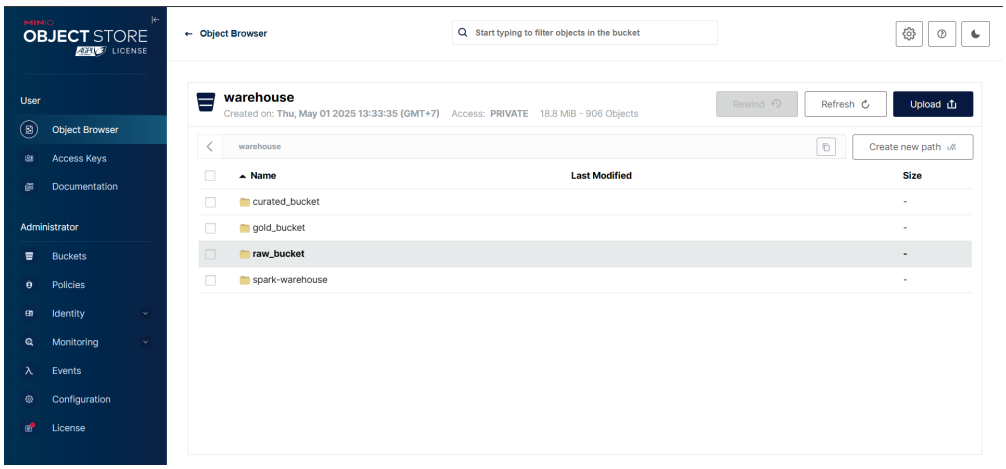

Figure 3.2: Raw Bucket destination

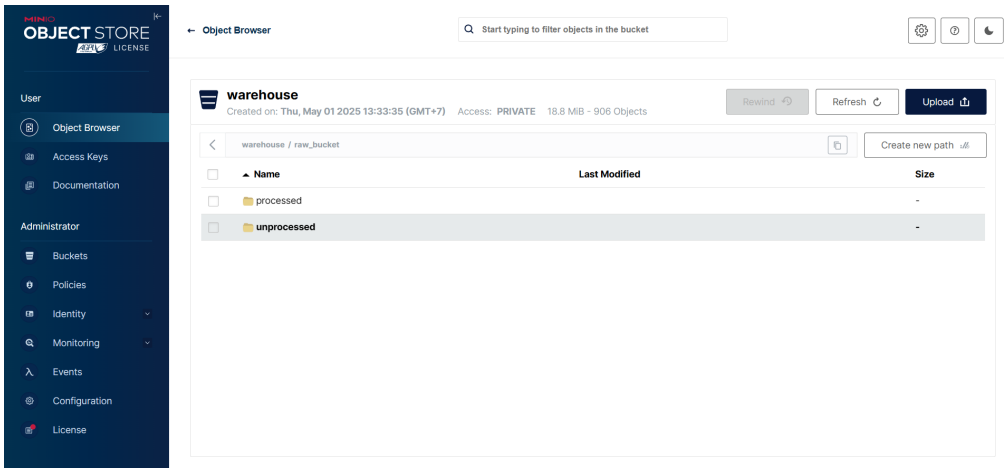

Figure 3.3: Path to Unprocessed folder

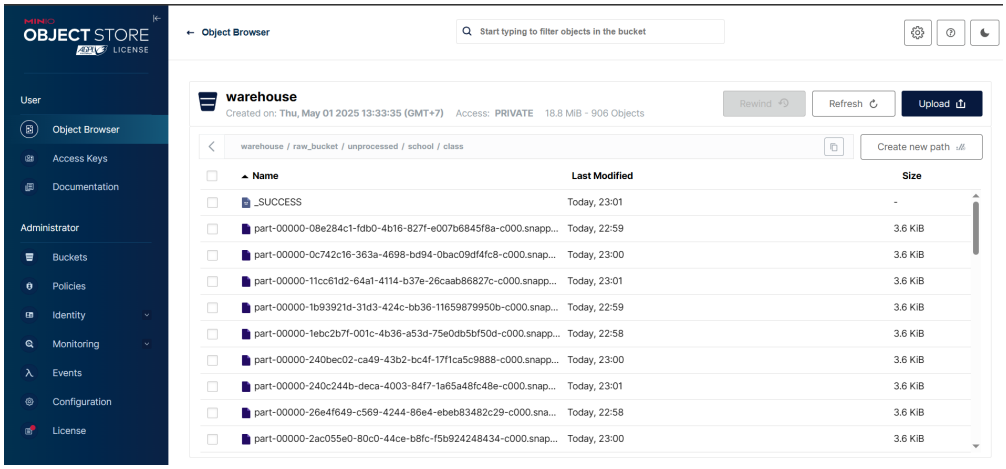

Figure 3.4: Raw data inside the path

3.2.3 Structured Data Processing with Apache Spark and Apache Hudi

Hudi Format Configuration in Spark Session

To transition data from the raw bucket to the silver bucket, Apache Hudi is integrated with Apache Spark. The Spark session is configured to support Hudi's write operations by including the necessary package dependencies and serialization settings. A minimal configuration is shown below:

Listing 3.2: Spark session initialization with Hudi support

```
spark = (
    SparkSession.builder
        .appName("Hudi Spark Example with MinIO")
        .config("spark.serializer", "org.apache.spark.serializer.
           KryoSerializer")
        .config("spark.jars.packages",
                "org.apache.hudi:hudi-spark3-bundle_2.12:0.15.0")
        .getOrCreate()
)
```

Once the session is initialized, Hudi write options are defined for each table to support upsert operations and schema synchronization with the Hive Metastore. These options include parameters for table naming, primary key configuration, precombine logic for deduplication, table type selection, and Hive sync details. An example configuration is as follows:

Listing 3.3: Hudi write options configuration

```
hudi_options = {
    'hoodie.table.name': table,
    'hoodie.datasource.write.recordkey.field': recordkey_str,
    'hoodie.datasource.write.table.name': table,
    'hoodie.datasource.write.operation': 'upsert',
    'hoodie.datasource.write.precombine.field': precombine_field,
    "hoodie.datasource.write.table.type": "COPY_ON_WRITE",
    "hoodie.datasource.hive_sync.enable": "true",
    "hoodie.datasource.hive_sync.mode": "hms",
    "hoodie.datasource.hive_sync.jdbcurl": "thrift://localhost:9083",
    'hoodie.datasource.hive_sync.database': database_name,
    'hoodie.datasource.hive_sync.table': table,
    'hoodie.datasource.hive_sync.support_timestamp': 'true',
}
```

This setup enables efficient, ACID-compliant data writes to MinIO in Hudi format, while automatically synchronizing the structured schema to Hive for querying purposes. Data written through this process is stored in the curated zone transforming unprocessed JSON

into queryable parquet files organized by table and partition structure.

Spark Job Configuration for Curated Storage Writing

The transformation and writing of structured data to curated storage is handled in the `write_to_curated_bucket.py` script. This Spark job is responsible for reading parsed JSON data from the `raw_bucket/unprocessed` folder in MinIO, transforming it according to predefined schemas, and writing the result as Apache Hudi tables into the `curated` bucket.

To support this, the script first configures the SparkSession with Hudi and MinIO integration:

Listing 3.4: Spark session configuration with Hudi and MinIO

```
def configure_spark_with_hudi_minio():
    spark = (SparkSession.builder
        .appName("Hudi Spark Example with MinIO")
        .config("spark.serializer", "org.apache.spark.serializer.
           KryoSerializer")
        .config("spark.jars.packages", "org.apache.hudi:hudi-spark3-
           bundle_2.12:0.15.0,")
        .config("spark.sql.extensions", "org.apache.spark.sql.hudi.
           HoodieSparkSessionExtension")
        .config("spark.sql.catalog.spark_catalog", "org.apache.spark.sql
           .hudi.catalog.HoodieCatalog")
        ...
        .enableHiveSupport()
        .getOrCreate())
    return spark
```

This configuration enables Spark SQL engine to interact with Hudi formatted tables and establishes connectivity with the MinIO object storage using the S3A protocol. Credentials, endpoints, and compatibility settings are all defined to allow Spark to read from and write to MinIO seamlessly.

Additionally, a Boto3 S3 client is initialized to enable programmatic access to MinIO for operations such as file inspection, deletion, or metadata management:

Listing 3.5: MinIO S3 client initialization with Boto3

```
def configure_minio_cursor():
    s3_client = boto3.client(
        "s3",
        endpoint_url="http://localhost:9008",
        aws_access_key_id="admin",
        aws_secret_access_key="password"
    )
    return s3_client
```

Together, these configurations establish the foundation for executing data transformation jobs that read unprocessed records and persist clean, structured datasets into the curated zone using Hudi's upsert and schema evolution features.

ETL Processing Logic for Curated Zone

After the Spark environment is configured, the ETL logic implemented in `write_to_curated_bucket` handles the transformation and load of clean data into the curated zone. This is achieved through three main functions: read_unprocessed_data, write_curated_data , and transfer_processed_data.

Reading Data from the Unprocessed Zone

Listing 3.6: Reading unprocessed data from MinIO

```
def read_unprocessed_data(spark_session, database, table):
    path = f"s3a://warehouse/raw_bucket/unprocessed/{database}/{table}"

    df = spark_session.read.parquet(path)

    for item in json_data:

        after_data = payload.get("after", {})
```

This function constructs the path to the raw input data stored in MinIO using the S3A protocol. It reads data in Parquet format from the `unprocessed` folder and extracts the JSON payloads. From each payload, only the `"after"` field is retained to represent the latest state of change data capture (CDC) events. If the schema is missing, the function returns an empty list.

Transforming and Writing Data to Hudi Tables

Listing 3.7: Writing data to Hudi curated zone

```
def write_curated_data(spark_session, data, database, table,
   primary_fields, precombine_field):

    for item in data:
        if "id" not in item:
            item["id"] = str(uuid.uuid4())

    df = spark.createDataFrame(data, schema=schema)
    df.write.format("hudi").options(**hudi_options).mode("append").save(
       data_path)
```

This function validates and transforms the JSON data into a structured DataFrame using inferred schema fields. It ensures the existence of primary keys (e.g., `id`) and fills in missing values. It then writes the data as a Hudi table into the curated MinIO bucket using the

COPY_ON_WRITE strategy. Key Hudi options enable upserts, precombine logic, Hive metastore synchronization, and timestamp support. This ensures that the curated layer maintains accurate, de-duplicated, and query-ready records.

Post-Processing: Moving Files to Processed Zone

Listing 3.8: Transferring processed data to avoid reprocessing

```
def transfer_processed_data(minio_cursor, database, table):

    minio_cursor.copy_object(...)
    minio_cursor.delete_object(...)
```

After the successful transformation and writing process, the processed files are moved from the `unprocessed` folder to the `processed` folder within the same MinIO bucket. This archival step avoids reprocessing of the same data during subsequent ETL runs, thereby maintaining pipeline idempotency.

Together, these functions form a lightweight and effective ETL workflow using PySpark, Apache Hudi, and MinIO. The design follows lakehouse principles by combining efficient storage with transactional updates and schema evolution features.

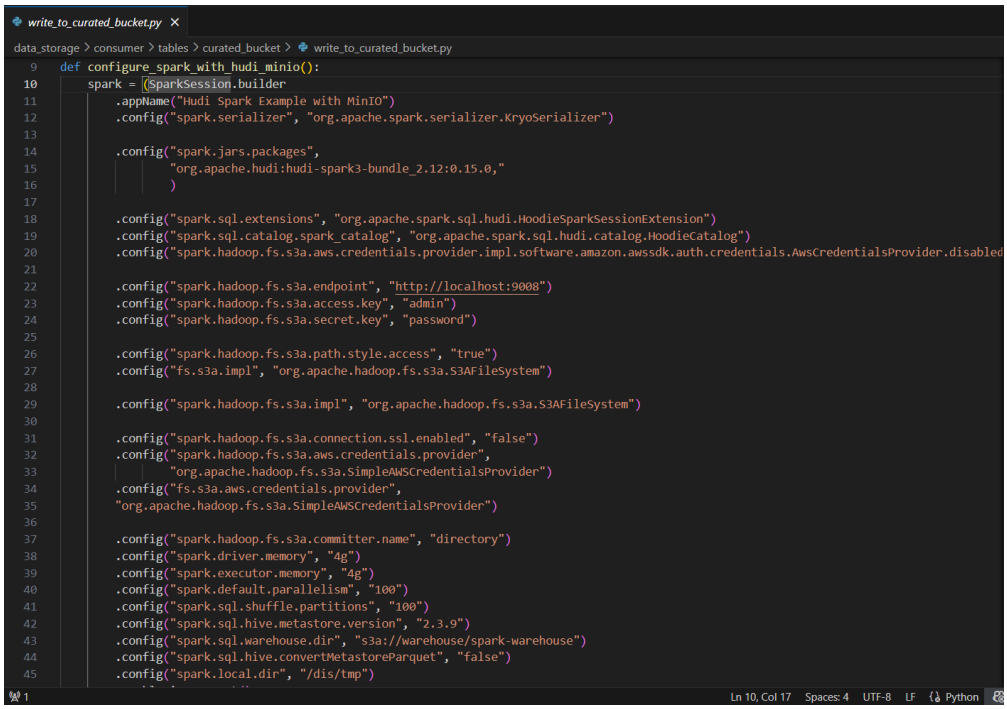

Figure 3.5: write_to_curated_bucket.py (1)

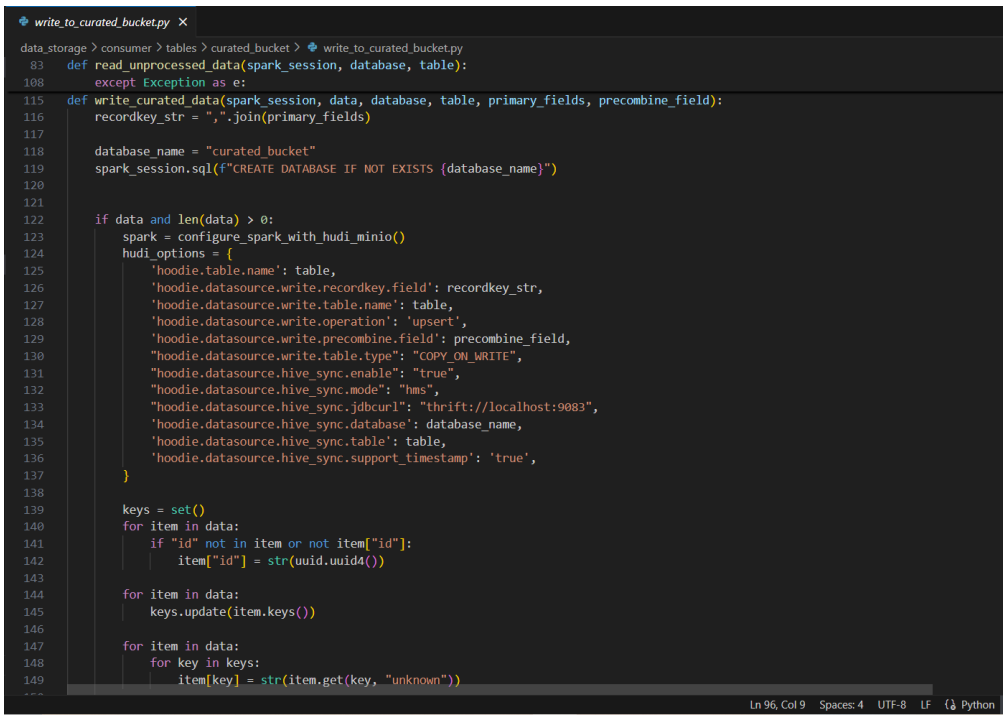

Figure 3.6: write_to_curated_bucket.py (2)

Write to Gold Bucket

The gold layer represents the final stage in the lakehouse architecture, where curated data is transformed into high-value, aggregated insights that serve as the foundation for analytical dashboards and business intelligence tools.

To persist this enriched data into the gold zone, the function `write_gold_data` is defined in the `write_to_gold_bucket.py` script. It encapsulates the logic for creating a Hudi table and writing aggregated data in a structured and queryable format.

Listing 3.9: Writing aggregated data into the gold layer

```
def write_gold_data(spark_session, folder_name, data, primary_field):
    database_name = "gold_bucket"
    spark_session.sql(f"CREATE DATABASE IF NOT EXISTS {database_name}")

    hudi_options = {
        'hoodie.table.name': folder_name,
        'hoodie.datasource.write.recordkey.field': primary_field,
        'hoodie.datasource.write.partitionpath.field': primary_field,
        ...
        "hoodie.datasource.write.table.type": "COPY_ON_WRITE",
        "hoodie.datasource.hive_sync.enable": "true",
        ...
    }

    data_path = f"s3a://warehouse/gold_bucket/{folder_name}"
```

```
    data.write.format("hudi").options(**hudi_options).mode("overwrite").
        save(data_path)
```

This function first ensures that the Hive-compatible database named `gold_bucket` exists. It then sets a series of Apache Hudi write options that control how the final dataset is persisted. In particular:

- `hoodie.datasource.write.operation` is set to `upsert`, enabling incremental updates to the gold tables.
- `hoodie.datasource.write.precombine.field` and `recordkey.field` are both set to a business-defined primary key, ensuring consistency in deduplication and ordering.
- `hoodie.datasource.hive_sync.enable` is activated to automatically sync the gold tables with the Hive Metastore for queryability using Spark SQL, Hive, or Trino.
- The output mode is defined as `overwrite`, which replaces the previous snapshot of the gold layer with the latest version of the computed metrics.

The output is written to MinIO at the `gold_bucket` path using the S3A protocol. This modular function is reusable across multiple downstream aggregation tasks and maintains strong data governance via Hive integration.

After being standardized in the curated layer, data undergoes transformation and aggregation to produce summary metrics that serve analytical and reporting purposes. These computations include total counts of students, staff, courses, classes, programs, and majors, as well as statistics such as graduation counts per term, average scores, and teaching loads.

The aggregation logic leverages Spark's `groupBy`, `agg`, and `countDistinct` functions. For example, the number of students per class is calculated as:

```
def students_by_class(students):
    return students.groupBy("class_code").agg(count("*").alias("
        total_students"))
```

In some cases, binary-encoded values such as student scores are decoded using a custom UDF before aggregation:

```
def avg_score_per_term(point):
    return point.withColumn("binary_point", unbase64("point_4")) \
                .withColumn("decoded_point", decode_decimal_udf("
                    binary_point")) \
                .groupBy("term_code") \
                .agg(round(avg("decoded_point"), 2).alias("
                    term_avg_score"))
```

The resulting summary data is written to the gold layer in Hudi format, enabling efficient access for dashboards and business intelligence tools.

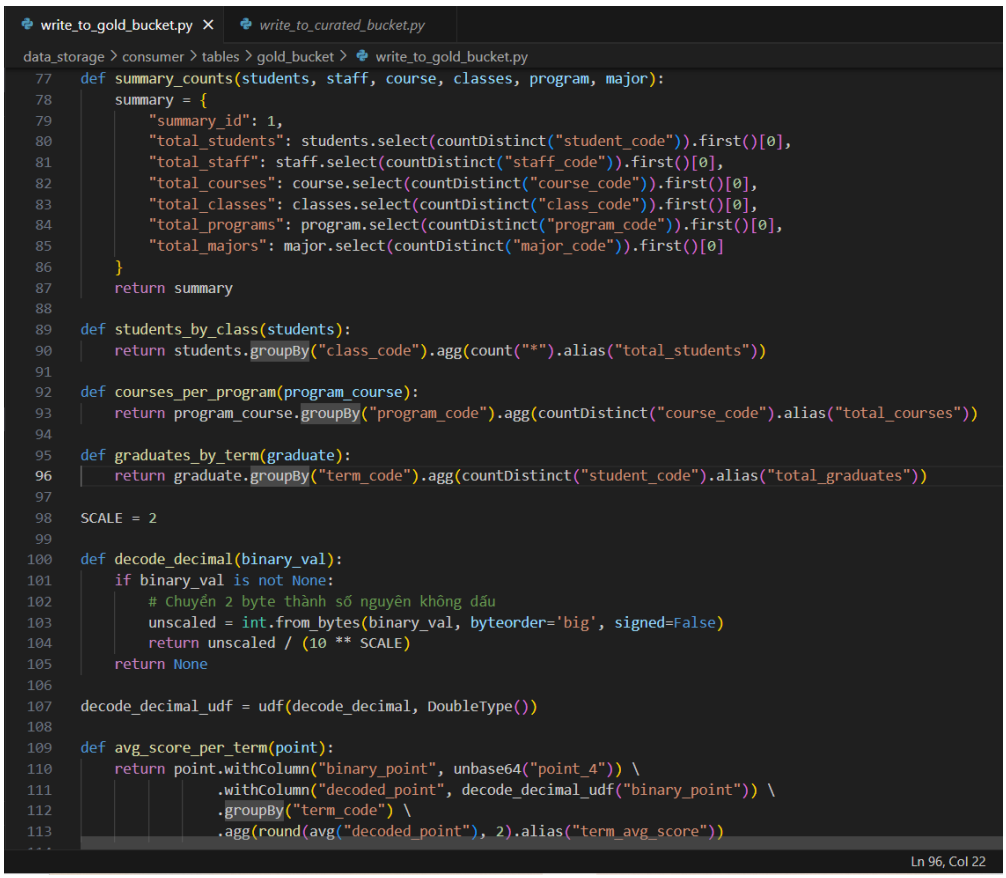

Figure 3.7: write_to_gold_bucket.py (1)

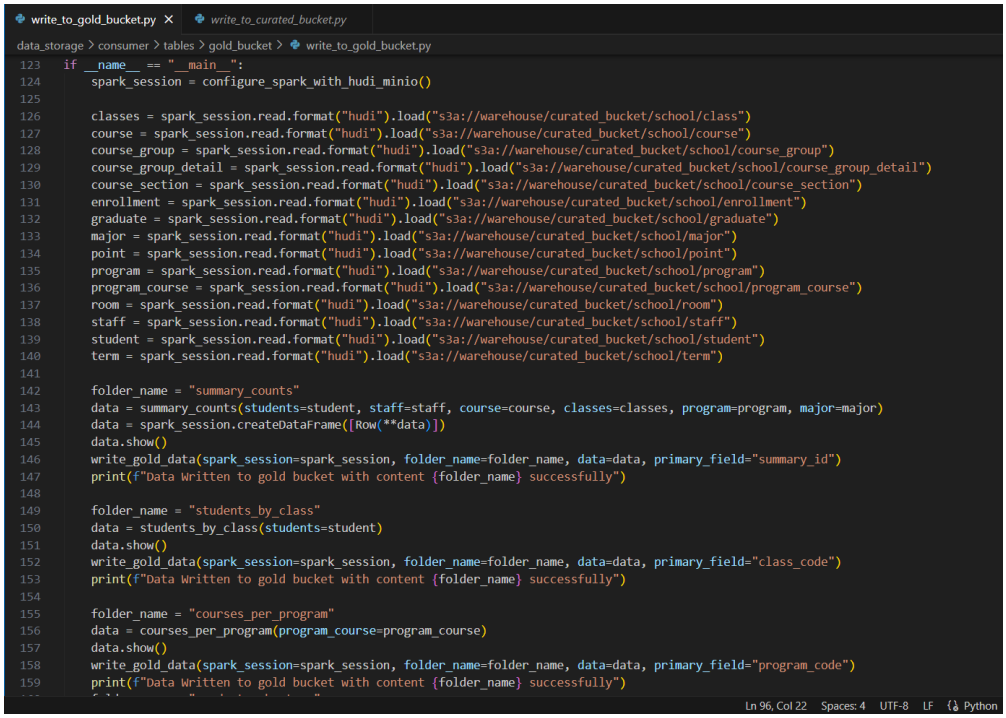

Figure 3.8: write_to_gold_bucket.py (2)

3.2.4 Orchestration with Airflow

We deployed Airflow locally, used to schedule and trigger our Spark jobs, relying on the default SQLite database to store Airflow metadata. In case of scaling system or applying in production environment, databases like PostgreSQL or MySQL will be considered as backend database.

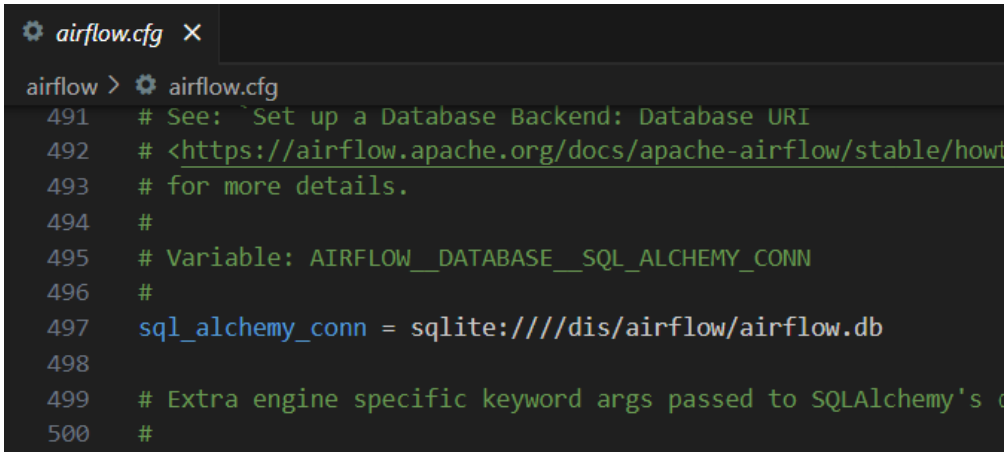

Figure 3.9: SQLite connection config

Simple Airflow DAG with a task that uses the BashOperator to activate a Python virtual environment and execute the datahub ingest command, which ingests metadata from Trino using a YAML configuration file.

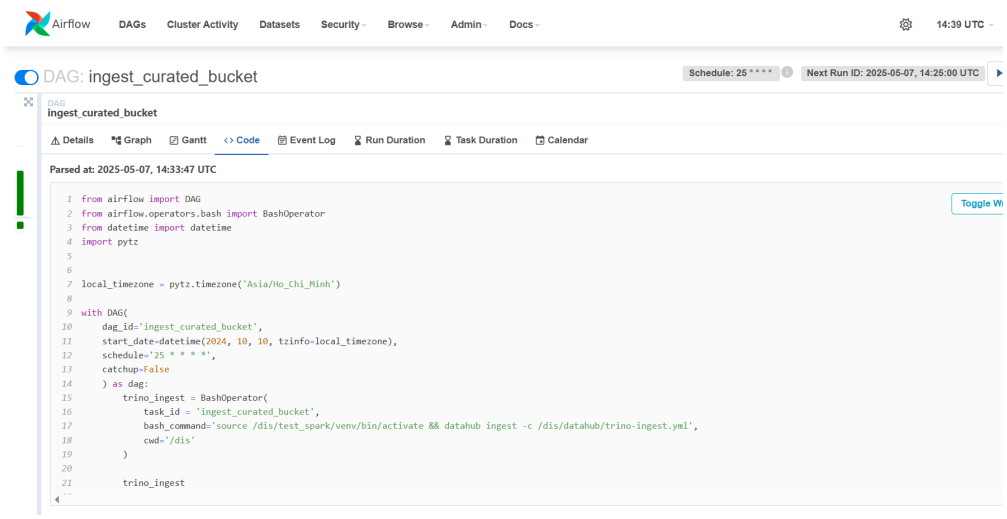

Figure 3.10: Sample Airflow dag

3.2.5 Query Engine: Trino Integration

Trino serves as the SQL query engine enabling federated querying over the Hudi datasets stored in the gold layer. Trino is deployed using a Docker container with a prebuilt image and linked to the Hive Metastore service via the HMS connector.

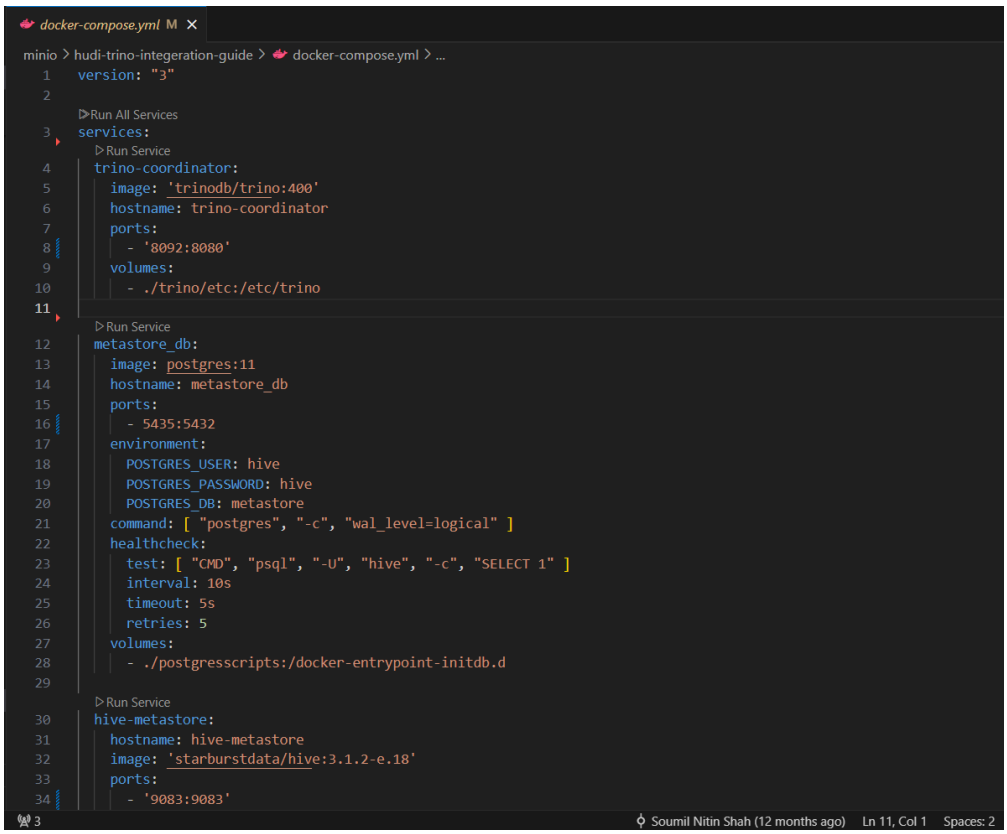

Figure 3.11: Trino and Hive Metastore Container

After the containers is up, we need to add the necessary configuration files to Trino server. Catalog configuration allow Trino to detect Hudi tables and connect with Hive Metastore:

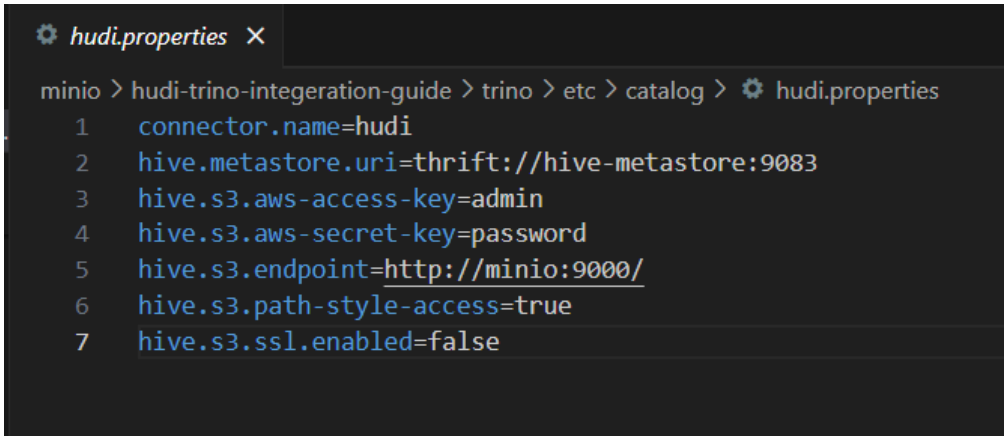

Figure 3.12: Trino Docker Container

Next, we configure the node.properties and config.properties files including node environment, coordinator settings and query parameters:

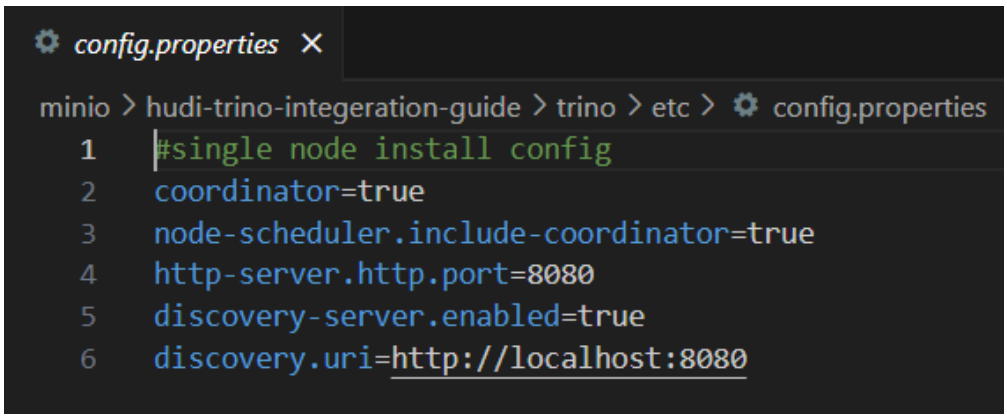

Figure 3.13: Trino Coordinator Config

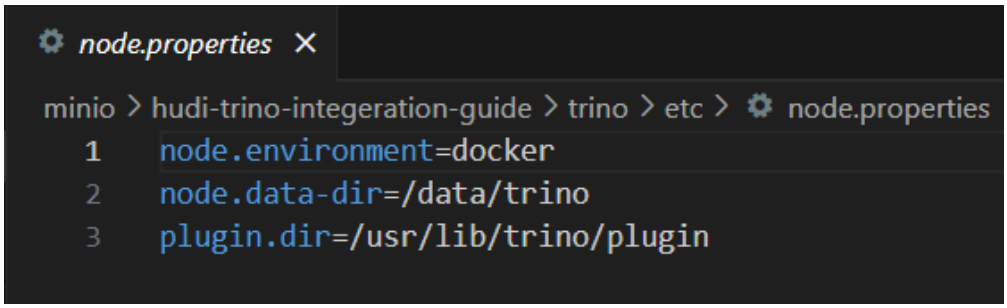

Figure 3.14: Trino Node Config

Once configured, the container is launched, and the Trino CLI can be used to query the datasets using commands for example:

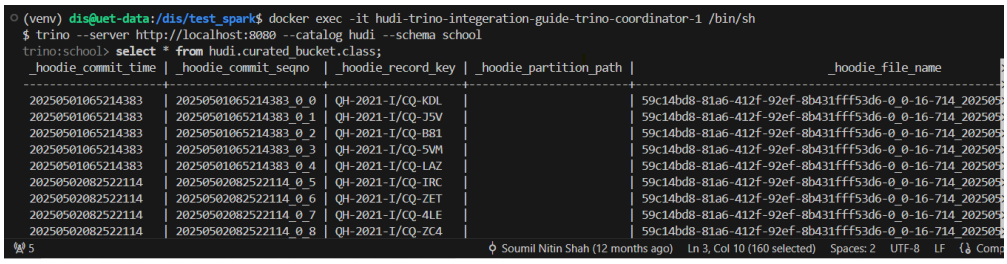

Figure 3.15: Trino CLI Command

3.2.6 Metadata Knowledge Graph: Datahub Integration

Datahub is also initialized using Docker, with standard services taken from the official documentation, modified to suit the system environment.

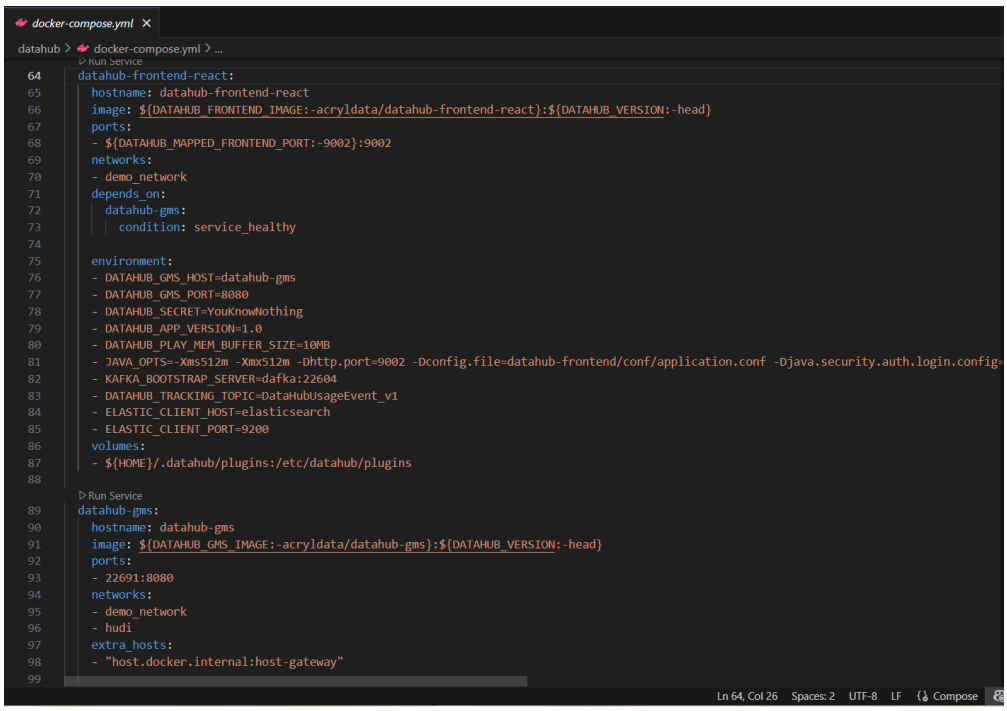

Figure 3.16: DataHub Docker Compose

In order for the Knowledge Graph to be able to collect metadata of the entire system, we need to ingest all services into Datahub, allowing Datahub to get the metadata of the services when the pipeline is launched. Datahub's ingestion feature can be used in both UI or running YAML files using CLI, depending on the support for each service.

Here we used UI Ingestion for MySQL and Kafka and CLI Ingestion for Trino and Metabase. The config includes information about host, port, api key or database.

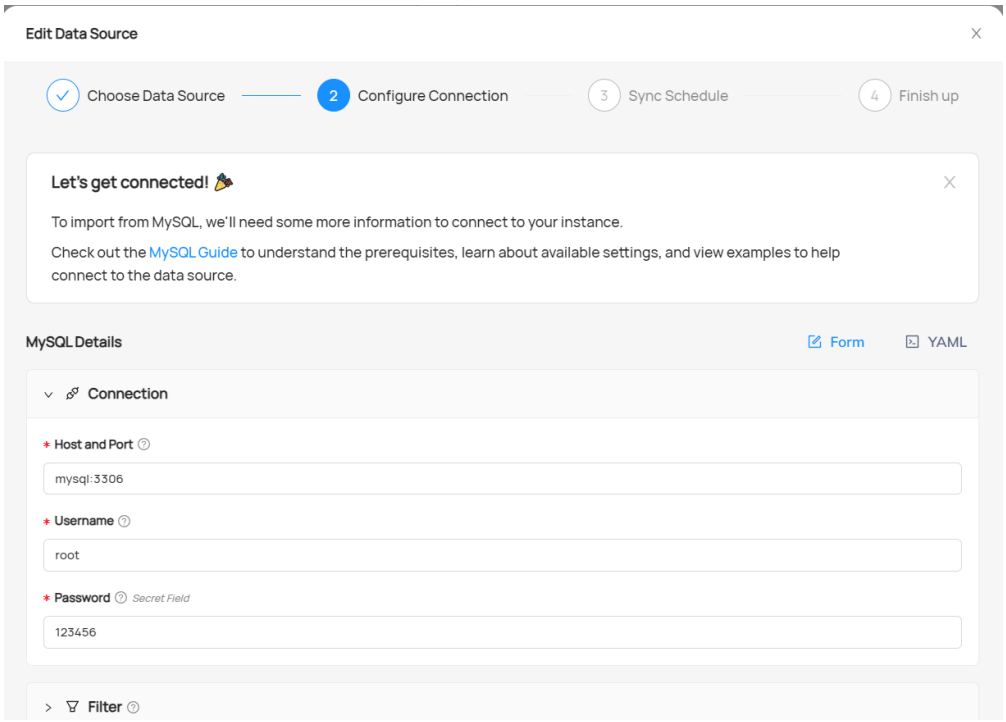

Figure 3.17: DataHub MySQL ingestion config

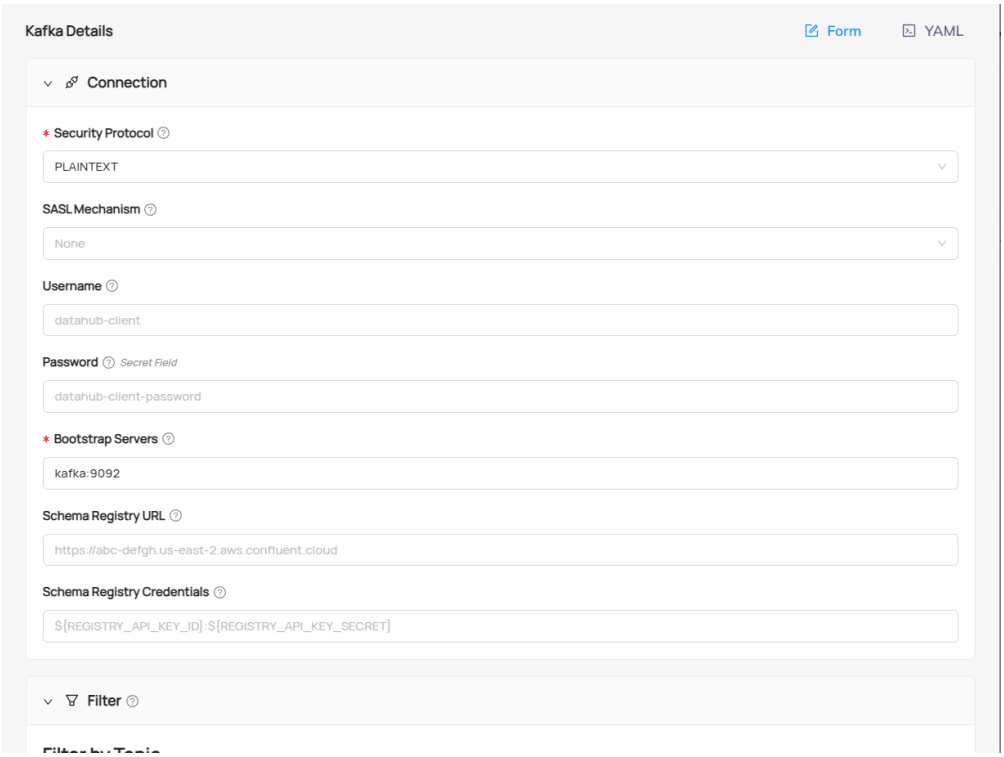

Figure 3.18: Datahub Kafka ingestion config

For Ingestion using CLI, we will need to run the following commands in the terminal:

```
$ datahub ingest -c trino-ingest.yml
$ datahub ingest -c metabase-ingest.yml
```

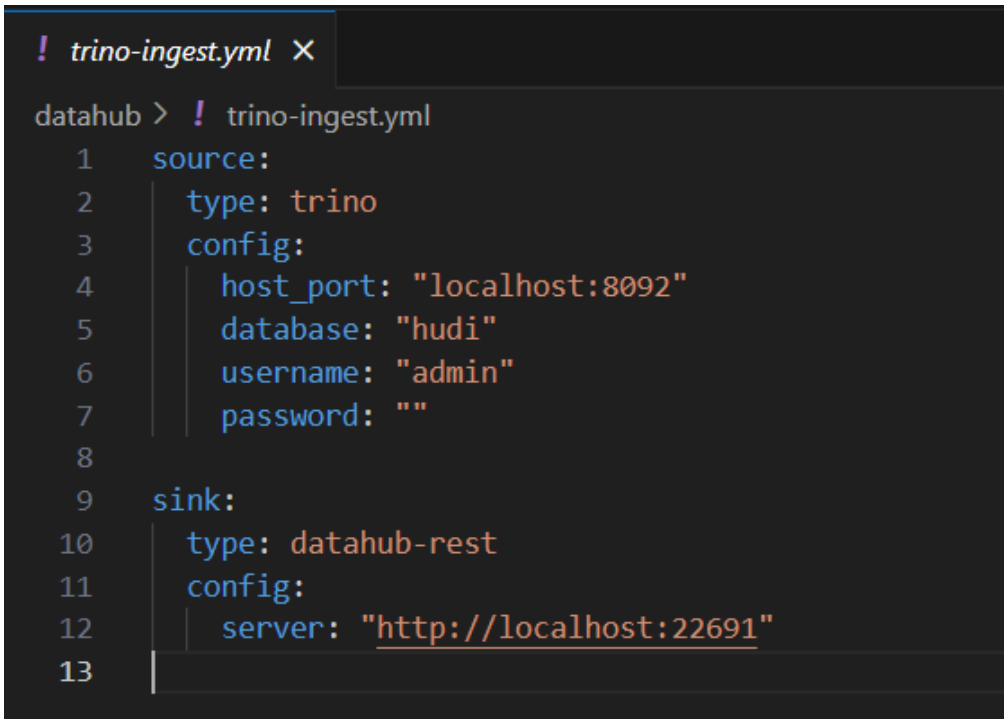

Figure 3.19: Datahub Trino ingestion config

```yaml
source:
  type: metabase
  config:
    connect_uri: 'http://localhost:3000'
    # api_key: mb_P9x0fSoUt43XrXtMVS+s9tXEJnVFF6AH1SZaJ+2xeUE= # All-users key not working (?)
    api_key: mb_pOi5zCLkCGtQjq3AnhalBTegjupx1zImT3nfnR2Ix4Y=

sink:
  type: datahub-rest
  config:
    server: "http://localhost:22691"
```

Figure 3.20: Datahub Metabase ingestion config

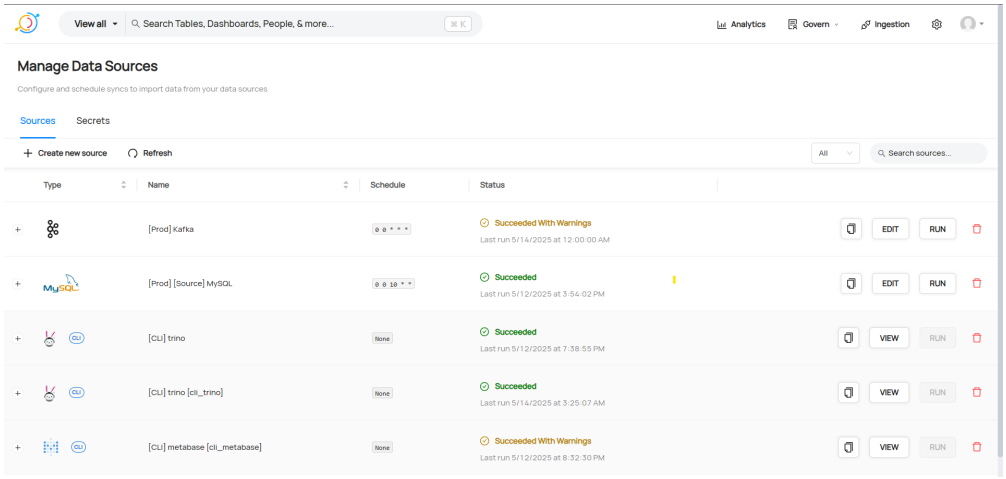

Figure 3.21: Datahub ingestion result

3.2.7 Metabase and LLM Integration

Metabase is deployed with Docker Compose, together with a PostgreSQL container used to store configuration and internal metadata.

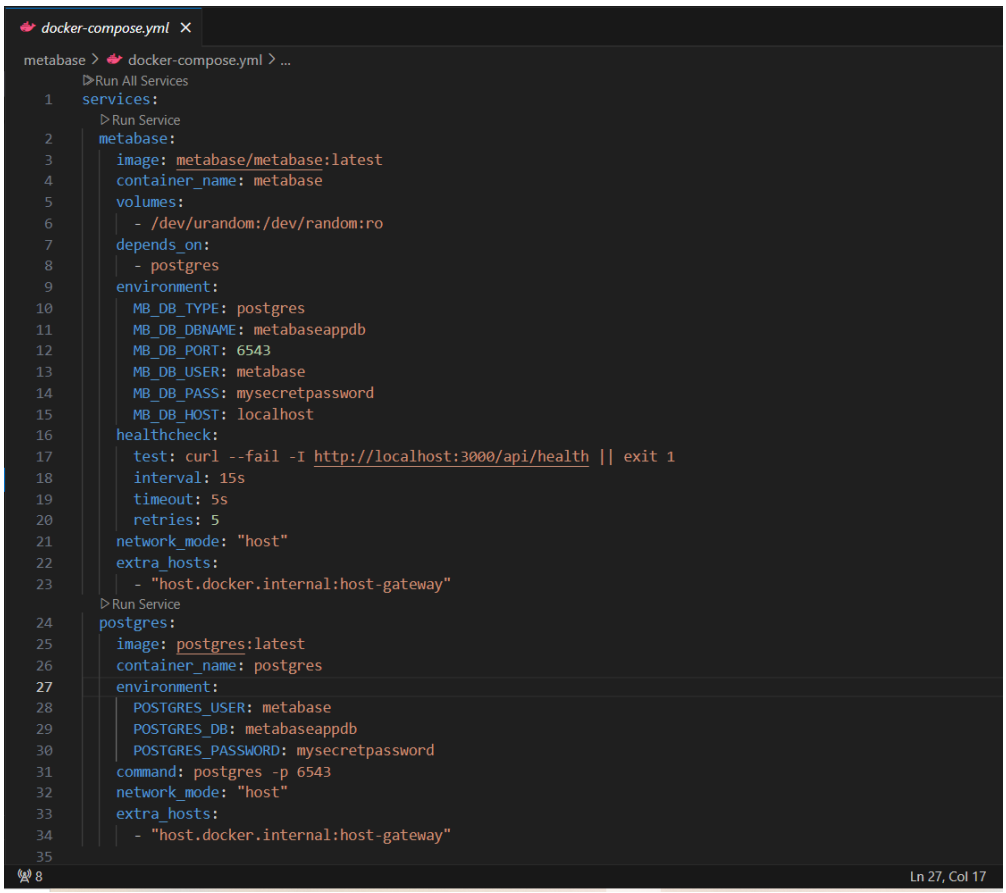

Figure 3.22: Metabase Docker Compose

After the Metabase container is up and running, we configure it to connect to Trino (Starburst Database type), which allows us to run queries, explore, and build dashboards on datasets stored in MinIO on Metabase UI.

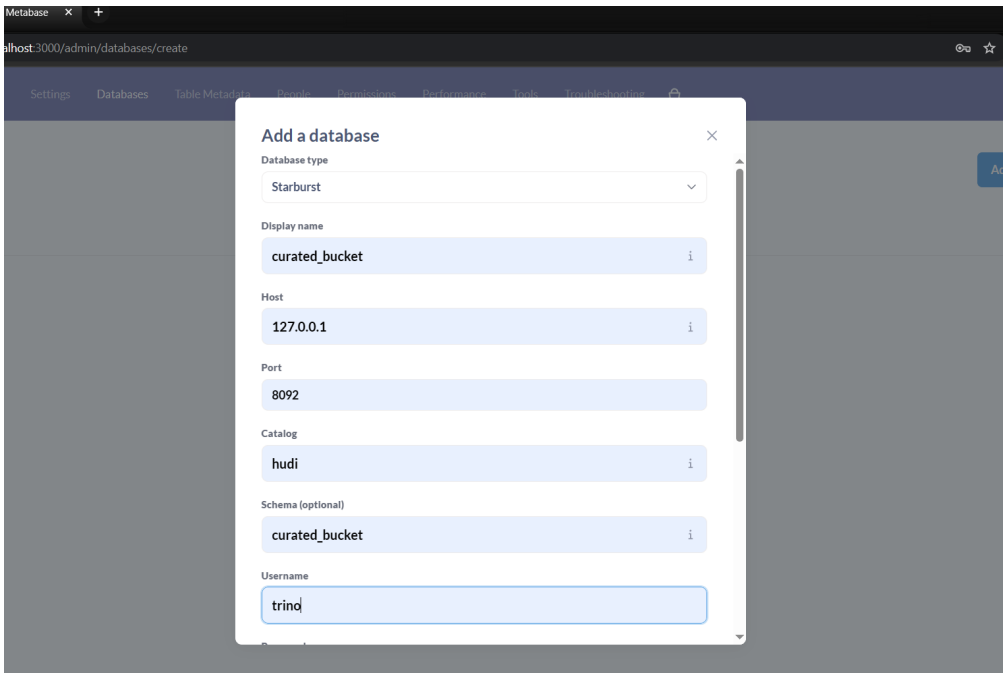

Figure 3.23: Metabase add database config

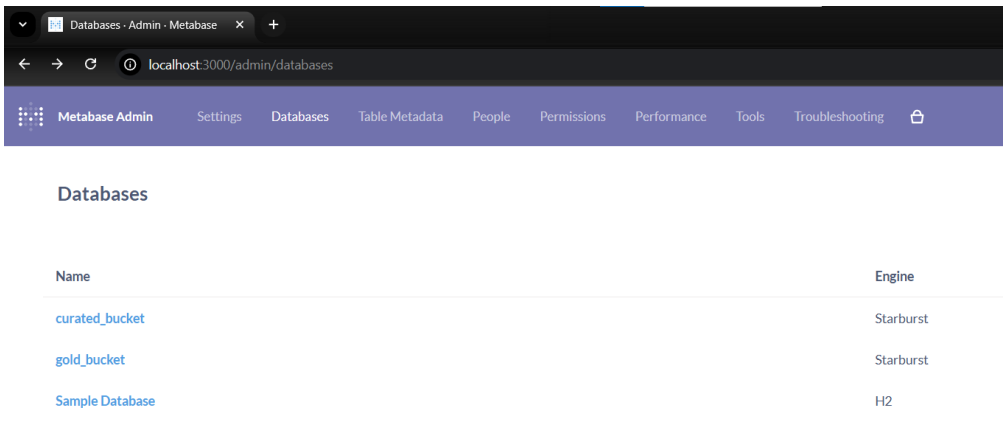

Figure 3.24: curated_bucket and gold_bucket in Metabase databases connection

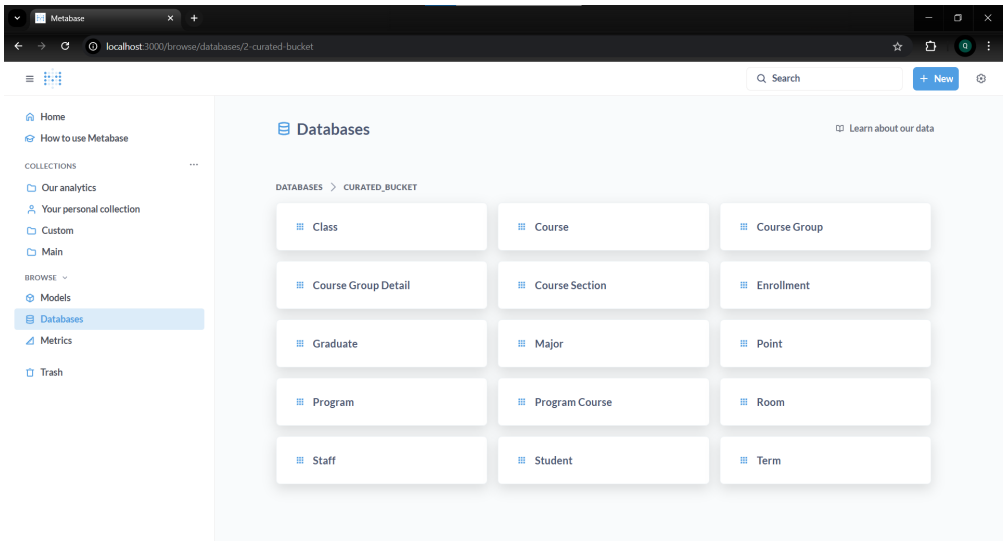

Figure 3.25: Curated Bucket Database

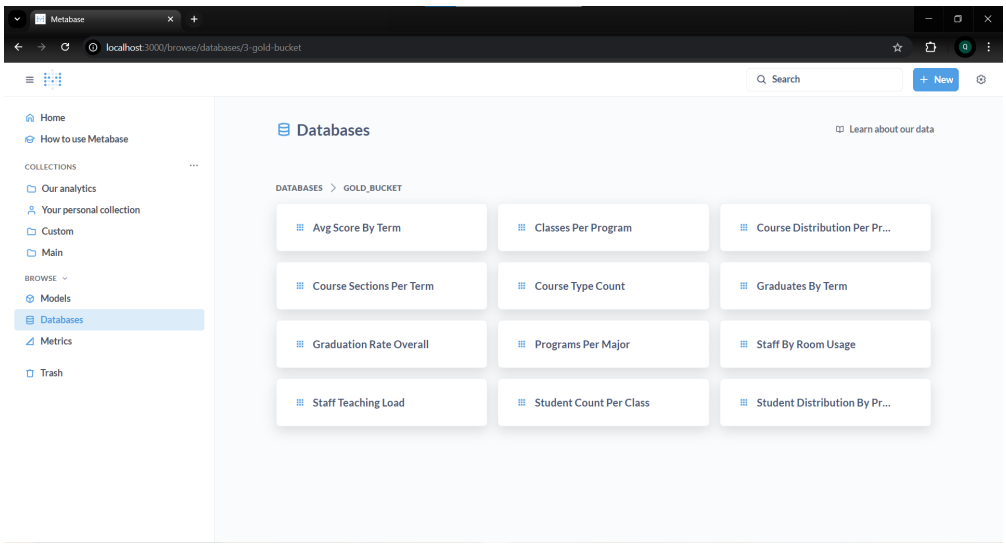

Figure 3.26: Gold Bucket Database

LLM-powered Module

To serve Intelligent Analytics, an LLM integrated application, specifically OpenAI's GPT-4o model, has been deployed by leveraging Metabase's API embedded in LLM's prompt so that the model is capable of generating dashboards based on idea descriptions from users.

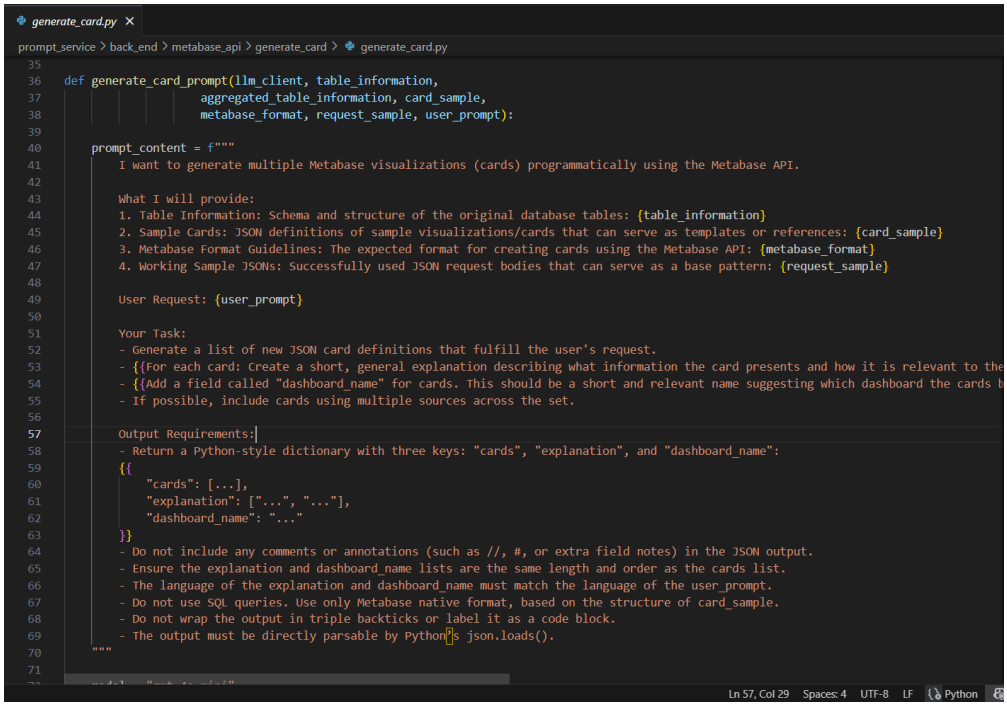

Figure 3.27: Predefined prompt to generate visualizations

generate_card.py | metabase_format.txt

prompt_service > back_end > metabase_api > prompt_data > metabase_format.txt

```
{
  "name": "Users by Month",
  "description": "Shows the number of new users grouped by month.",
  "collection_id": 1,
  "display": "bar",
  "dataset_query": {
    "type": "question",
    "query": {},
    "database": 1                // ID of the database
  },
  "visualization_settings": {
    "graph": {
      "type": "bar",             // Chart type: "bar", "line", "pie", "table", etc.
      "x-axis": "month",         // Field used for x-axis
      "y-axis": "total"          // Field used for y-axis
    }
  },
  "collection_position": null,   // Optional; controls order in the collection
  "parameters": [],              // Optional;
  "param_values": []             // Optional; used to pre-fill parameter values
}

```

Figure 3.28: Format Config for Metabase API

Whenever a request comes in from a user, the application converts it to a prompt so that LLM can generate JSON strings containing information for the visualizations. These JSON pieces are then sent to Metabase's API to generate the dashboard on the UI.

create_dashboard.py

prompt_service > back_end > metabase_api > generate_card > create_dashboard.py

```python
import requests

BASE_URL = "http://localhost:3000/api/dashboard"
API_KEY = "mb_qtEhHV7Xn7+7T08gq5kGESrWaLNeHvYdDHM0kBshRws="


headers = {
    "Content-Type": "application/json",
    "X-API-Key": API_KEY
}


def create_dashboard(name):
    dashboard_payload = {
        "collection_id": 7,
        "name": name,
        "cache_ttl": 1,
        "description": "",
    }

    res = requests.post(BASE_URL, json=dashboard_payload, headers=headers)
    res.raise_for_status()
    return res.json()

```

Figure 3.29: Post request to generate dashboard

3.3 Results and Evaluation

3.3.1 Scenario: Academic performance analysis of the university through the end-to-end process

This scenario manages a complete end-to-end data pipeline to provide an overview of academic performance, supporting analysis by automatically generating visual charts based on user prompts to gain insights about the university.

The dataset consists of 15 tables provided by the operations department of the University of Engineering and Technology - Vietnam National University, Hanoi. The dataset revolves around information about students, majors, scores, graduation, terms, courses.

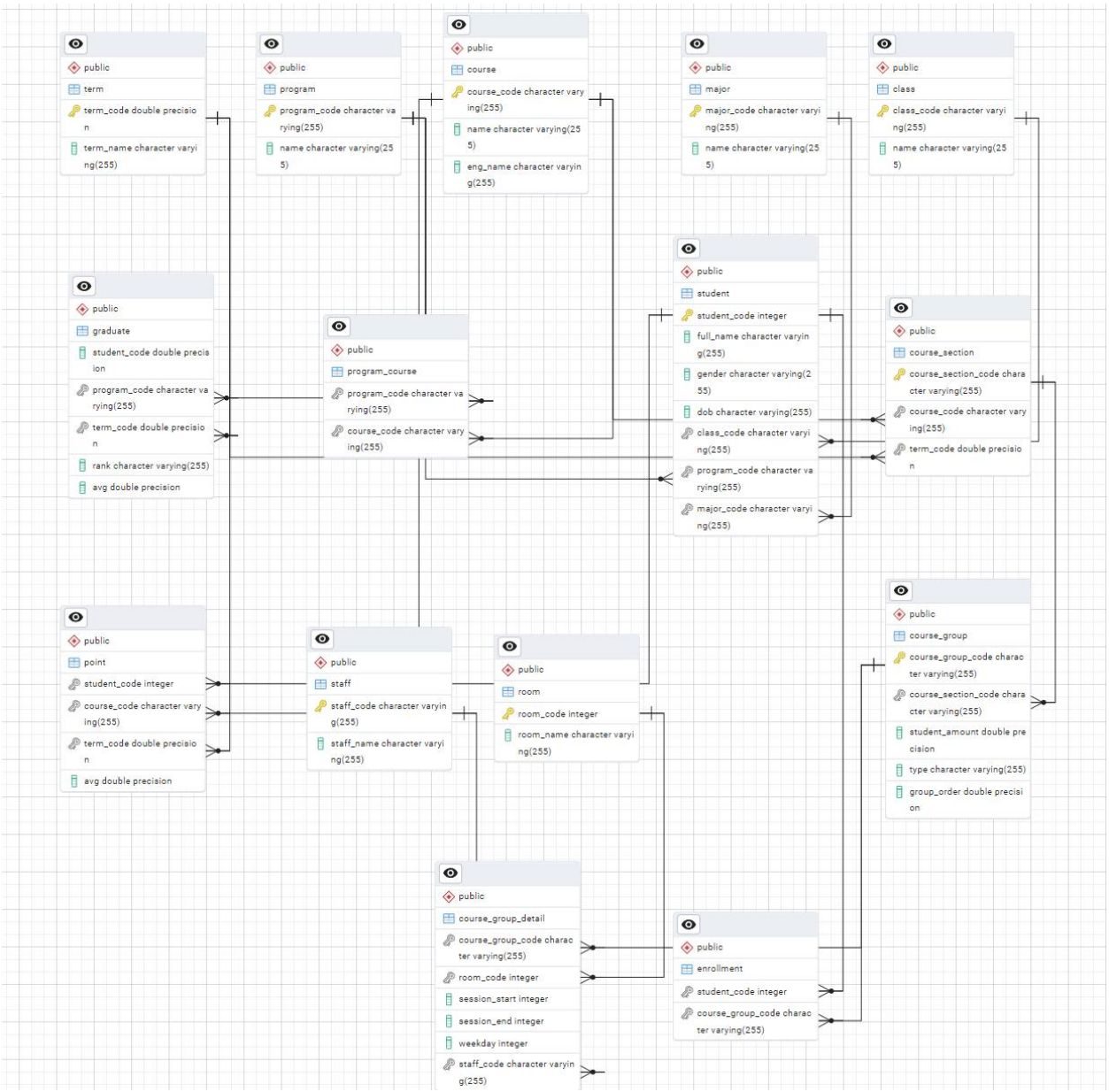

Figure 3.30: Dataset Schema used for Scenario

As usual, new data are updated in the data source, specifically MySQL in this case, and

changes are captured by Debezium to be sent to a Kafka topic, which is then accessed by Kafka consumers and forwarded to the raw bucket stored in MinIO.

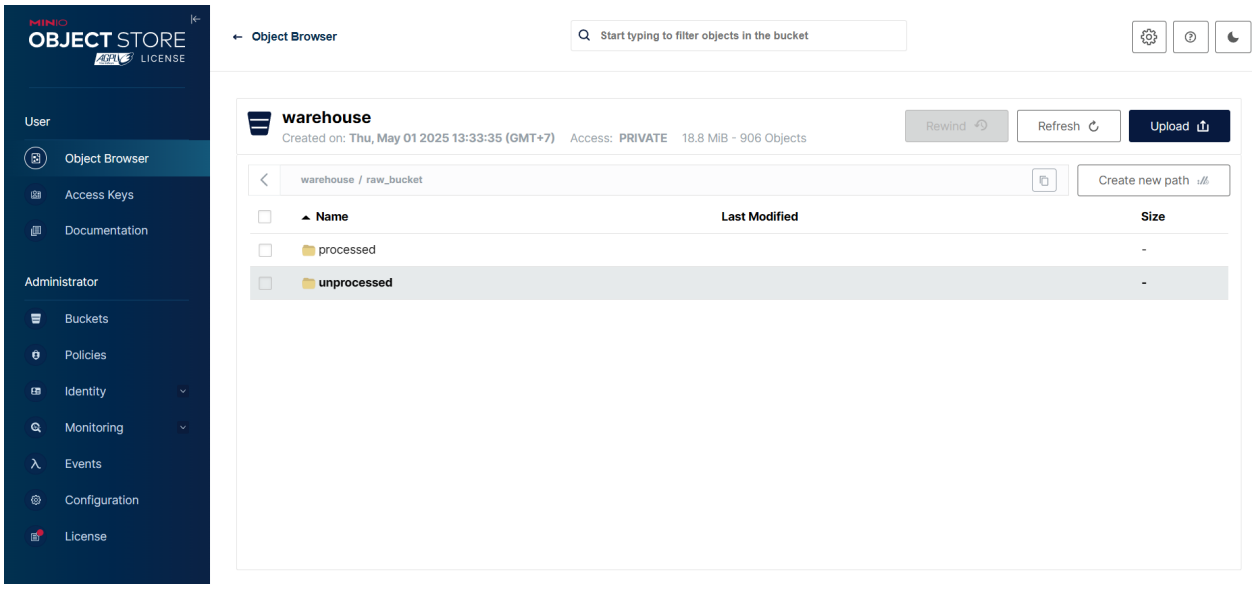

Figure 3.31: New data captured by Debezium and Kafka

With this overview scenario, we would like to ingest new data from all available tables. As shown in the image, it takes only about 4 seconds to transfer 100,000 records from 15 tables.

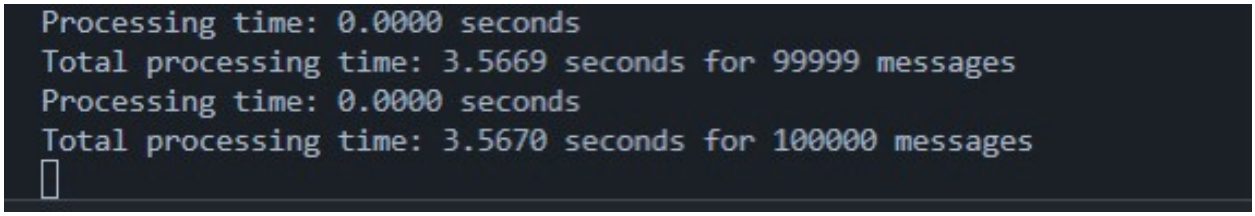

Figure 3.32: Time to sync 100,000 records

Next, we sequentially run Spark jobs to write the new data in Hudi table format into the curated bucket, and then aggregate it based on various metrics for analysis purposes, storing the results in the gold bucket.

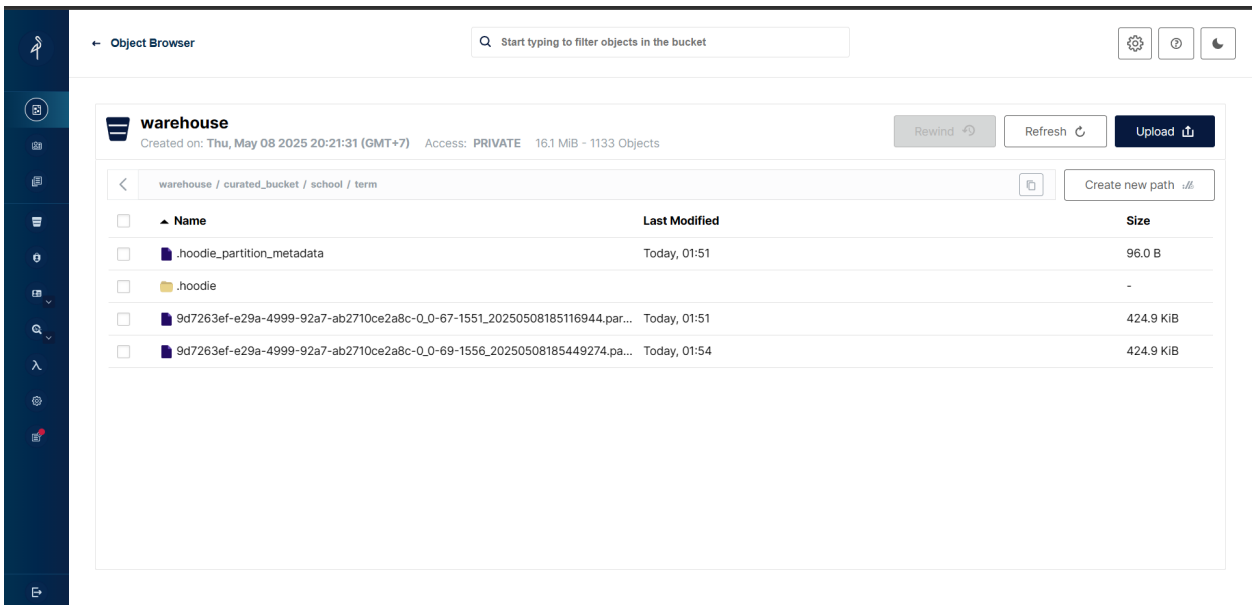

Figure 3.33: New data in curated bucket

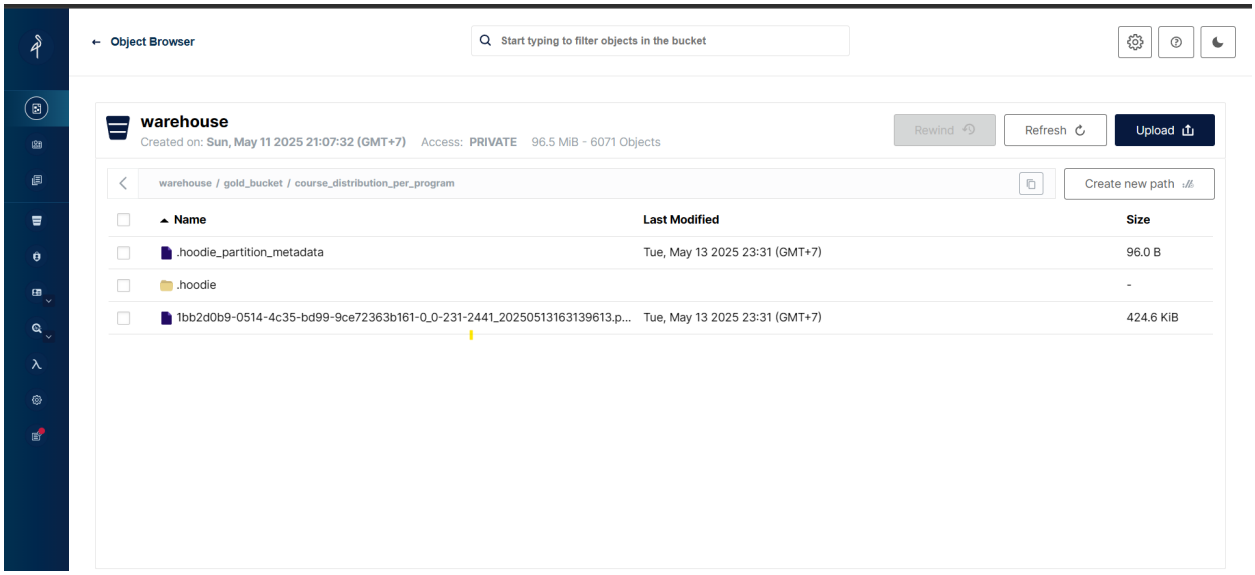

Figure 3.34: Aggregated data in gold bucket

After the data are aggregated into the gold bucket, users can send prompts to the LLM engine via API endpoint or Prompt UI to generate dashboards based on their requests-in this case-specifically analyzing scores and the number of graduates by academic term.

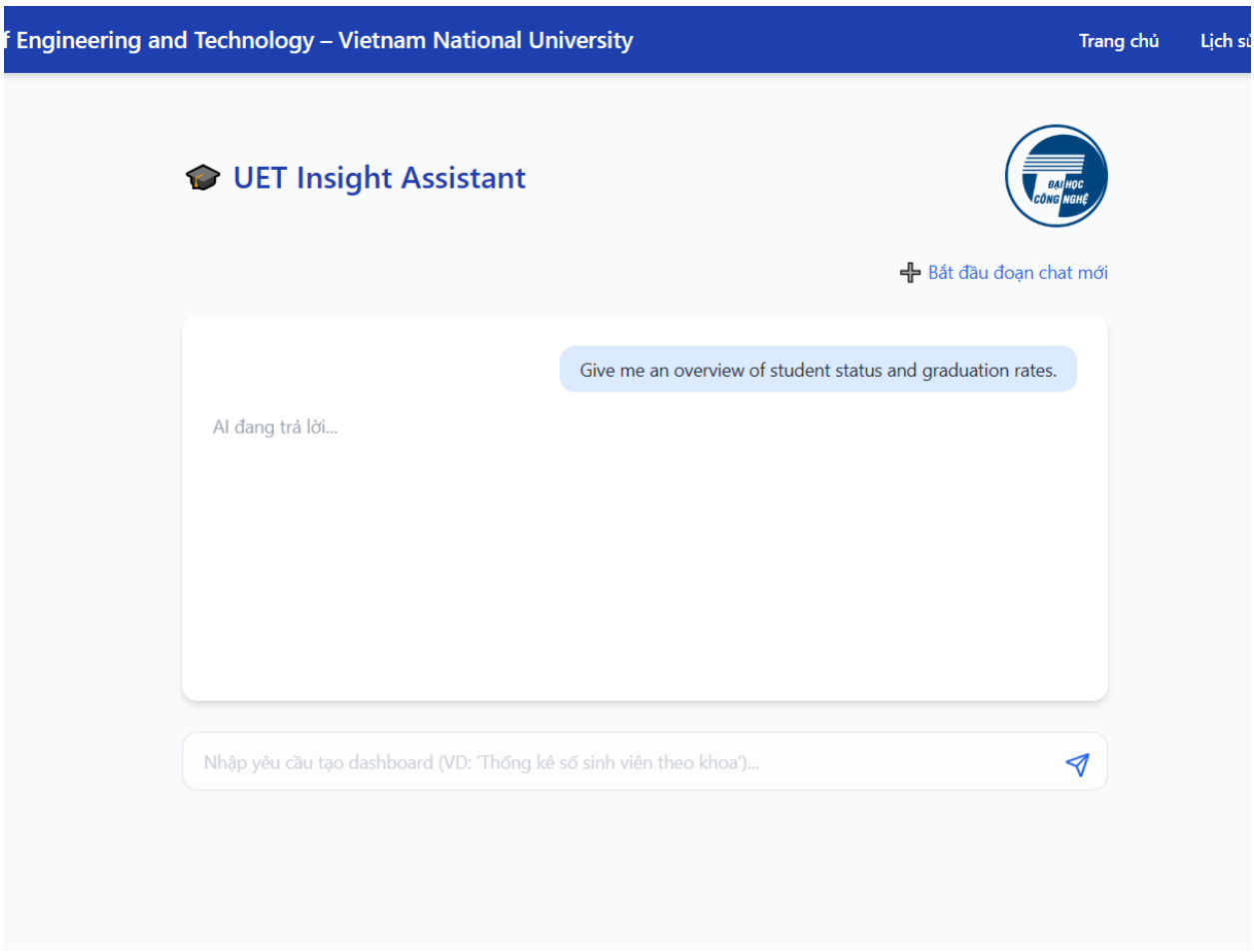

Figure 3.35: Prompt to gen dashboard

Some visualizations use basic metrics which are already available in the Gold Bucket such as **Graduation By Term, Average GPA** while others use SQL query to aggregate like **Student Amount By Course**

```
def graduates_by_term(graduate):
    return graduate.groupBy("term_code").agg(countDistinct("student_code").alias("total_graduates"))

SCALE = 2

def decode_decimal(binary_val):
    if binary_val is not None:
        unscaled = int.from_bytes(binary_val, byteorder='big', signed=False)
        return unscaled / (10 ** SCALE)
    return None

decode_decimal_udf = udf(decode_decimal, DoubleType())

def avg_score_per_term(point):
    return point.withColumn("binary_point", unbase64("point_4")) \
                .withColumn("decoded_point", decode_decimal_udf("binary_point")) \
                .groupBy("term_code") \
                .agg(round(avg("decoded_point"), 2).alias("term_avg_score"))
```

Figure 3.36: Metrics to aggregate graduates_by_term and avg_score_per_term

Figure 3.37: SQL query generated by LLM engine

The pipeline running in the system is also updated in real-time on the Knowledge Graph in Datahub, allowing us to manage changes to each event that occurs. We can add labels to entities using Datahub Metadata Annotation feature (automatic or manual), which enhances the semantics of the chart for alerting or impact analysis tasks.

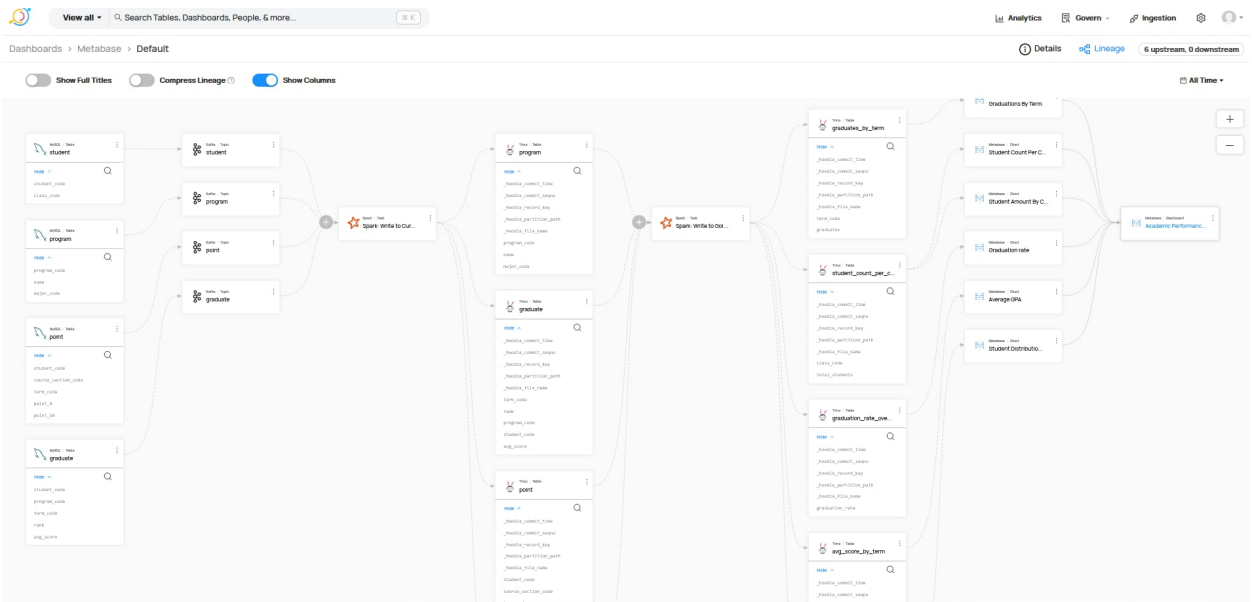

Figure 3.38: Metadata Knowledge Graph in Datahub

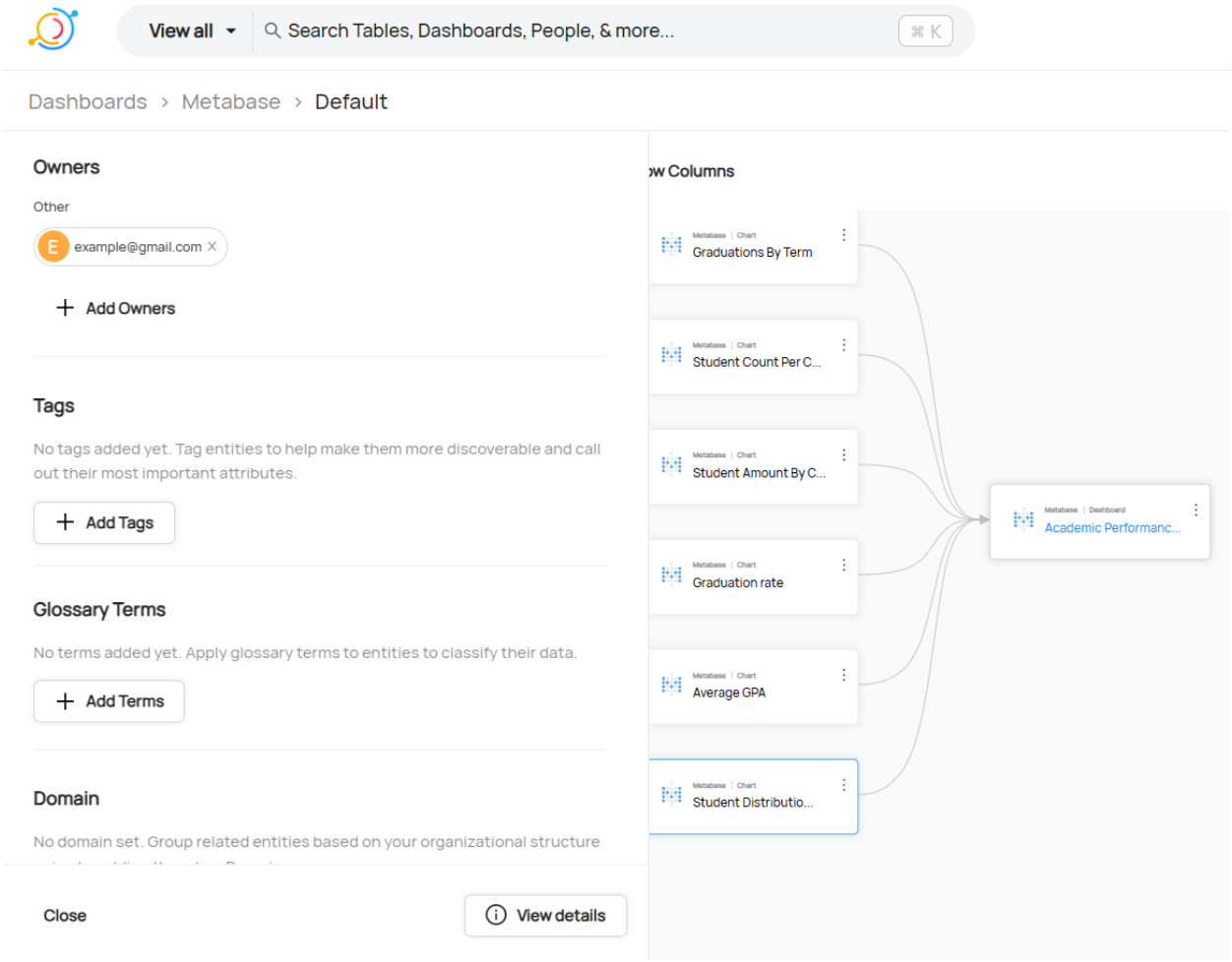

Figure 3.39: Add labels to a visualization like owner

The dashboard is generated by the LLM using Metabase's API, with visualizations tailored to the predefined analytical goals.

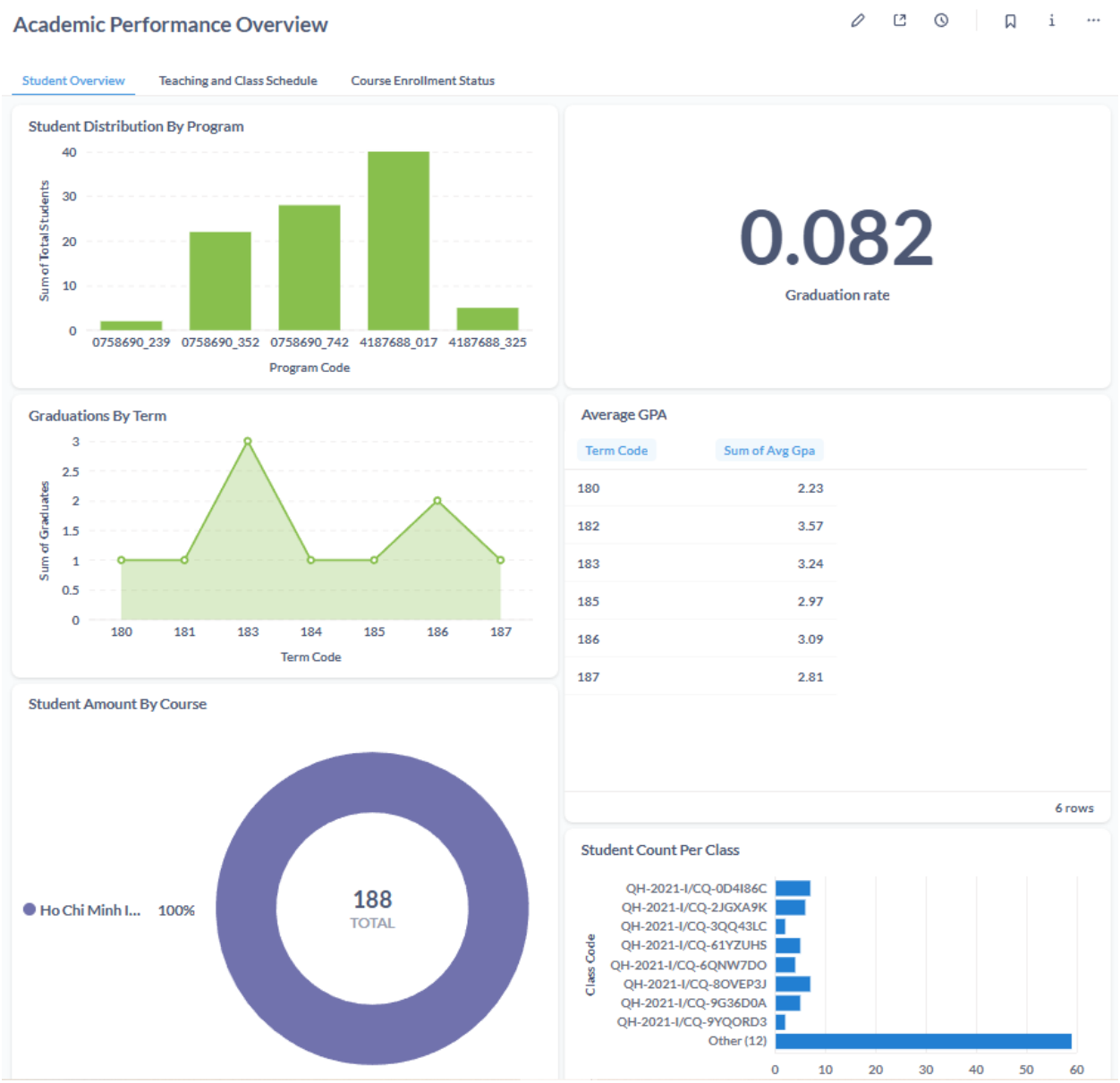

Figure 3.40: Final Dashboard

3.3.2 Evaluation

The experiment demonstrates the system's capability to handle large volumes of data in a short time, while also supporting end users in data analysis and visualization through the integrated LLM module. It successfully meets the defined requirements and shows potential for handling more complex business scenarios.

3.4 Chapter Summary

This chapter has presented the integration steps of various services within the system, along with a test scenario to validate performance and problem-solving capabilities in real-world business use cases. As a result, the thesis has demonstrated both the theoretical

foundation and practical effectiveness of the proposed system architecture.

CONCLUSION AND PERSPECTIVES

Main Contributions

In this thesis, we have implemented a metadata knowledge graph for a modern data platform integrated with intelligent analytics. The main contributions of the study include:

- Gaining a comprehensive understanding of knowledge graph theories, intelligent analytics integration, and modern architectural approaches to standard data platforms.
- Proposing a unified system architecture in which various open-source components are utilized to build a functional and scalable platform.
- Successfully implementing the proposed system using open-source technologies and ensuring seamless integration across all core services.
- Conducting testing and evaluation of the platform through specific, practical scenarios and an university dataset.

Thesis Limitations

Despite receiving valuable guidance from the supervisor, the scope of the platform remains constrained due to limited experience and time. Several limitations that we expect to improve in the near future:

- LLM module does not cover all cases when handling analysis or visualization requests from users.
- Knowledge graph does not automatically add labels for entities, requiring manual action from the user.
- LLM module is not yet integrated to learn the semantics of knowledge graphs, graph-related analysis currently has to be found and inferred by users themselves.

Future Work

With further guidance, the platform will be incrementally optimized to address current shortcomings and improve its analytical capabilities. Future directions include:

- Enabling automation across the entire Knowledge Graph, serving impact analysis
- Integration so that LLM can understand the semantics of knowledge graphs, improving the quality of intelligent analysis
- Improve the performance of LLM, aiming to solve a complex analysis request with full steps from query to visualization.

Bibliography

[1] Metabase overview. URL https://www.metabase.com/. Accessed: 2025-04-26.

[2] Chatgpt by openai. URL https://openai.com/chatgpt. Accessed: 2025-04-26.

[3] Trino architecture. URL https://trino.io/docs/current/overview/architecture.html. Accessed: 2025-04-26.

[4] Apache Kafka Project. Apache kafka documentation - introduction. https://kafka.apache.org/intro, 2024. Available at: https://kafka.apache.org/intro.

[5] Apache Software Foundation. What is airflow®? https://airflow.apache.org/docs/apache-airflow/stable/index.html. Accessed: 2025-04-26.

[6] Apache Software Foundation. Apache spark components. https://spark.apache.org/docs/latest/, 2024. Accessed: 2025-04-26.

[7] Yihan Chen, Xin Luna Dong, and Sen Wu. A survey on large language models for data analytics. arXiv preprint arXiv:2307.08594, 2023. URL https://arxiv.org/abs/2307.08594.

[8] DataHub Project. Datahub components overview, 2024. URL https://datahubproject.io/docs/components/. Accessed: 2025-04.

[9] Debezium Project. Debezium documentation (v2.5). https://debezium.io/documentation/, 2024. Available at: https://debezium.io/documentation/.

[10] Apache Software Foundation. Apache hudi documentation. https://hudi.apache.org/docs/overview, 2024. Accessed: 2025-04-26.

[11] Claas Giebler, Jan-Peter Ostberg, and Norbert Ritter. Leveraging modern data pipelines for real-time data integration. In Datenbanksysteme für Business, Technologie und Web (BTW), pages 393–412, 2019.

[12] Google. The knowledge graph: About, 2012. URL https://www.google.com/search/about/. Accessed: 2025-04-18.

[13] Google. Introducing the knowledge graph: things, not strings, 2012. URL https://blog.google/products/search/introducing-knowledge-graph-things-not/. Accessed: 2025-04-20.

[14] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Axel Polleres, Fabian M. Suchanek, Rudi Studer, and Antoine Zimmermann. Knowledge Graphs. Morgan & Claypool Publishers, 2021. doi: 10.2200/S01125ED1V01Y202109WBE021.

[15] IDC. The global datasphere: Forecast to 2025. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf, 2021. Accessed: 2025-04-18.

[16] Microsoft Corporation. Sql server 2022 documentation. https://learn.microsoft.com/en-us/sql/sql-server/, 2024. Available at: https://learn.microsoft.com/en-us/sql/sql-server/.

[17] Inc. MinIO. Minio documentation. https://docs.min.io, 2023. Accessed: 2025-04-26.

[18] Oracle Corporation. Mysql 8.0 reference manual. https://dev.mysql.com/doc/refman/8.0/en/, 2024. Available at: https://dev.mysql.com/doc/refman/8.0/en/.

[19] Hoa NN Quoc Anh. Building a modern data platform based on data fabric architecture, 2019.

[20] Microsoft Research. Semantic analytics: The next evolution of data intelligence, 2022. URL https://www.microsoft.com/en-us/research/project/semantic-analytics. Accessed: 2025-04-20.

[21] Microsoft Research. Graphrag: Unlocking llm discovery on narrative private data, 2024. URL https://www.microsoft.com/en-us/research/blog/graphrag-unlocking-llm-discovery-on-narrative-private-data/. Accessed: 2025-04-20.

[22] Amazon Science. Building commonsense knowledge graphs to aid product recommendation, 2024. URL https://www.amazon.science/blog/building-commonsense-knowledge-graphs-to-aid-product-recommendation. Accessed: 2025-04-20.

[23] Amazon Web Services. Lakehouse with apache hudi and amazon s3. https://aws.amazon.com/blogs/big-data/building-an-open-lakehouse-using-apache-hudi-on-amazon-s3/, 2022. Accessed: 2025-04-26.

[24] Wayne Xin Zhao et al. A survey of large language models for data analytics: Opportunities and challenges. arXiv preprint arXiv:2306.05087, 2023.