---------------------------------------------------------------Report on Metadata Management in Modern Data Architectures

---------------------------------------------------------------

Submitted by:

Subash Khanal

Data Intern

Submitted to:

eXtensoData

Submitted on:

27th May, 2025

This report is submitted as a partial fulfillment of the internship requirement at eXtensoData under the Department of Computer Science and Engineering, Kathmandu University.

**TABLE OF CONTENT**

[LIST OF TABLES 2](#_Toc199365010)

[Chapter 1 Executive Summary 3](#_Toc199365011)

[1.1 Purpose 3](#_Toc199365012)

[1.2 Key Findings 4](#_Toc199365013)

[1.3 Recommendations 6](#_Toc199365014)

[Chapter 2 Introduction 10](#_Toc199365015)

[2.1 Background 10](#_Toc199365016)

[2.2 Objectives 10](#_Toc199365017)

[Chapter 3 Metadata Management Overview 12](#_Toc199365018)

[3.1 Definition and Importance 12](#_Toc199365019)

[3.2 Types of Metadata 13](#_Toc199365020)

[3.2.1 Business Metadata 13](#_Toc199365021)

[3.2.2 Technical Metadata 13](#_Toc199365022)

[3.2.3 Operational Metadata 14](#_Toc199365023)

[Chapter 4 Tools for Metadata Management 15](#_Toc199365024)

[4.1 Installation and Configuration 15](#_Toc199365025)

[4.1.1 Tool 1: Apache Atlas 15](#_Toc199365026)

[4.1.2 Tool 2: DataHub 17](#_Toc199365027)

[4.2 Challenges in Tool Installation 23](#_Toc199365028)

[4.2.1 Challenges in Installation : Apache Atlas 23](#_Toc199365029)

[4.2.2 Challenges in Installation : DataHub 24](#_Toc199365030)

[Chapter 5 Capabilities Comparison 26](#_Toc199365031)

[5.1 Comparative Analysis Framework 26](#_Toc199365032)

[5.2 Comparative Matrix 28](#_Toc199365033)

[5.3 Analysis of Strengths and Weaknesses 31](#_Toc199365034)

[Chapter 6 Metadata Ingestion Workflow 33](#_Toc199365035)

[6.1 Workflow Overview 33](#_Toc199365036)

[6.2 Task Breakdown 34](#_Toc199365037)

[Chapter 7 Findings and Analysis 62](#_Toc199365038)

[7.1 Summary of Findings 62](#_Toc199365039)

[7.2 Challenges in Metadata Management 62](#_Toc199365040)

[Chapter 8 Conclusion 64](#_Toc199365041)

[References 65](#_Toc199365042)

[Appendices 66](#_Toc199365043)

**LIST OF FIGURES**

[Figure 6.1. 1 DAG for metadata ingestion in DataHub 33](#_Toc199185601)

[Figure 6.1. 2 DAG for metadata ingestion in Apache Atlas 34](#_Toc199185602)

[Figure I. 1 Entity Detail for Account with its classification…………………………………. 49](#_Toc199185609)

[Figure I. 2 Summary of Data Classifications and their counts 49](#_Toc199185610)

[Figure I. 3 Customer Sensitive Data Classification Dashboard 50](#_Toc199185611)

[Figure I. 4 Entity Type Distribution and Activity Status 50](#_Toc199185612)

[Figure I. 5 Business Metadata Management Interface 51](#_Toc199185613)

[Figure I. 6 Financial Transaction Search Results Interface 51](#_Toc199185614)

[Figure I. 7 Financial Glossary Management Interface 52](#_Toc199185615)

[Figure I. 8 Transaction Relationships Data Lineage View 52](#_Toc199185616)

[Figure I. 9 Financial Transaction Metadata View 53](#_Toc199185617)

[Figure I. 10 Type System Network Visualization 54](#_Toc199185618)

[Figure II. 1 DataHub - PostgreSQL Platform Data Catalog Overview……………………… 54](#_Toc199185619)

[Figure II. 2 Custom View Creation Dialog 55](#_Toc199185620)

[Figure II. 3 Dataset Schema View with Field-Level Metadata 55](#_Toc199185621)

[Figure II. 4 Dataset Documentation Tab View 56](#_Toc199185622)

[Figure II. 5 Dataset Properties Configuration View 57](#_Toc199185623)

[Figure II. 6 SQL Queries Tab with Highlighted Query Examples 57](#_Toc199185624)

[Figure II. 7 Glossary management interface for HR term group 58](#_Toc199185625)

# LIST OF TABLES

[Table 5.I. 1 General Comparison Apache Atlas Vs DataHub 28](#_Toc199187175)

[Table 5.II. 1 Core Metadata Features Comparison Apache Atlas Vs DataHub………………29](#_Toc199187184)

[Table 5.III. 1 Data lineage and Impact Analysis Comparison Apache Atlas Vs DataHub… ..30](#_Toc199187192)

[Table 5.IV. 1 Search and Discoverablilty Comparison Apache Atlas Vs DataHub……………30](#_Toc199187197)

[Table 5.V. 1 Community and EcoSystem Comparison Apache Atlas Vs DataHub……….......31](#_Toc199187204)

# Chapter 1 Executive Summary

This report evaluates metadata management solutions through Proof of Concept (POC) implementations of Apache Atlas, DataHub, and Amundsen, focusing on data governance, lineage tracking, and discovery within modern data architectures. The POCs utilized a sample employee data and sample financial transaction data processing use case, integrating with Apache Airflow, Apache Spark, Apache Hive (for Atlas), and PostgreSQL (for DataHub). Apache Atlas and DataHub successfully demonstrated robust metadata ingestion, lineage visualization, and governance capabilities, with Atlas excelling in governance-heavy environments and DataHub offering user-friendly ingestion and scalability. Amundsen’s installation was successful, but metadata ingestion failed across multiple attempts, and the UI partially loaded without displaying sample metadata, contrary to official documentation expectations. Investigations point to Elasticsearch indexing issues, configuration errors, and potential UI rendering bugs as root causes.

## 1.1 Purpose

The purpose of this report is to analyze the capabilities, performance, and challenges of Apache Atlas, DataHub, and Amundsen for metadata management, with a specific focus on diagnosing and resolving Amundsen’s metadata ingestion and UI issues. The report aims to:

1. **Evaluate Tool Capabilities**: Assess Atlas and DataHub’s metadata ingestion, lineage, and governance features, and explore Amundsen’s discovery potential despite ingestion and UI failures.
2. **Diagnose Amundsen Issues**: Investigate metadata ingestion failures and partial UI loading in Amundsen, using recent findings and official documentation to identify causes and solutions.
3. **Support Decision-Making**: Inform stakeholders on tool selection, emphasizing Atlas and DataHub’s strengths and Amundsen’s potential post-resolution.

## 1.2 Key Findings

The POCs for Apache Atlas, DataHub, and Amundsen provide insights into their metadata management capabilities, with detailed investigation into Amundsen’s ingestion and UI issues based on prior conversations, official documentation, and web sources.

**1.2.1 Apache Atlas**

1. **Strengths**:
2. **Robust Metadata Model**: Supports flexible entity types, classifications, and business metadata, enabling governance for employee data processing in Hive.
3. **Comprehensive Lineage**: End-to-end lineage visualization (CSV → Spark → Hive) supports impact analysis and compliance, viewable in the Atlas UI (<http://localhost:21000).>
4. **Governance Features**: PII tagging and schema capture ensure compliance, as demonstrated in the POC.
5. **Integration**: Seamlessly integrates with Airflow, Spark, and Hive via the Atlas Hive hook and REST API, capturing metadata automatically.
6. **Extensible API**: Facilitates custom integrations for enterprise workflows.
7. **Challenges**:
8. **Performance at Scale**: Graph queries and indexing may slow with large metadata volumes, requiring optimization.
9. **Integration Complexity**: Custom integrations demand expertise due to limited advanced documentation.
10. **User Experience**: UI can be slow for complex lineage graphs; search functionality needs improvement.

**1.2.2 DataHub**

1. **Strengths**:
2. **User-Friendly Interface**: Web UI simplifies metadata exploration and search, enhancing accessibility for non-technical users.
3. **Flexible Ingestion**: Dynamic URNs and APIs streamline metadata capture, with glossary term associations for fields like Employee ID and Base Salary.
4. **Governance Features**: Automated PII/Sensitive tagging, ownership assignment, and lineage tracking enhance governance.
5. **Challenges**:
6. **Dependency Management**: Startup order issues require depends\_on and health checks for stability.
7. **URN Precision**: Incorrect URNs (e.g.,urn:li:dataset:(urn:li:dataPlatform:postgres,etl\_pipeline.employees,PROD) prevent metadata visibility, requiring exact matches.

**1.2.3 Amundsen**

1. **Strengths**:
2. **Discovery Potential**: The Flask/React frontend with Neo4j and Elasticsearch offers intuitive data discovery when fully functional, ideal for self-service analytics.
3. **Flexible Ingestion**: Databuilder supports custom data loading, with sample data scripts provided for testing.
4. **Modular Architecture**: Independent microservices (Frontend, Metadata, Search) allow flexible deployment.
5. **Challenges**:
6. **Metadata Ingestion Failures**: Multiple ingestion attempts (e.g.,sample\_neo4j\_data\_loader.py, custom CSV loading) failed, with no metadata appearing in Neo4j or Elasticsearch. Common issues include:
   * + - **Elasticsearch Indexing Errors**: Metadata loaded into Neo4j but not indexed in Elasticsearch, often due to low memory (error 137) or configuration mismatches (e.g., incorrect key formats).
       - **Connection Issues**: Neo4j connection refusals (ConnectionRefusedError: [Errno 111]) due to incorrect credentials or port conflicts (7474/7687).
       - **Databuilder Issues**: ModuleNotFoundError for databuilder module, resolvable by installing in the virtual environment (pip install .) or setting PYTHONPATH.
7. **UI Problems**: The UI partially loads and fails to display sample metadata, contrary to Amundsen’s official documentation ([https://www.amundsen.io/amundsen/).](https://www.amundsen.io/amundsen/).%20) Potential causes include:

* Elasticsearch indexing failures preventing metadata visibility.
* UI rendering bugs, possibly due to JavaScript errors or misconfigured endpoints in frontend/configs.
* Insufficient Docker memory (default 2GB) causing container crashes.

1. **Complex Setup**: Local setups require manual Neo4j/Elasticsearch configuration, increasing complexity.
2. **Limited Governance**: Focuses on discovery, lacking advanced lineage or classification compared to Atlas.

## 1.3 Recommendations

Based on the POC findings and Amundsen’s matured status, the following recommendations focus on Apache Atlas and DataHub, emphasizing their best use cases, scenarios where they excel, and areas where they are less suitable. Amundsen’s prior use cases are noted for context but excluded from primary recommendations due to its reduced adoption and unresolved issues in the POC.

**1.3.1 Apache Atlas: Best Use Cases and Limitations**

1. **Best Use Cases**

* **Governance-Heavy Environments**: Ideal for organizations in regulated industries (e.g., finance, healthcare) requiring strict compliance, PII tagging, and detailed lineage tracking. The POC demonstrated robust governance with employee data in Hive, capturing schema, PII classifications, and lineage (CSV → Spark → Hive).
* **Hadoop Ecosystem Integration**: Excels in environments using Hive, Spark, and HBase, leveraging the Atlas Hive hook for automatic metadata capture and lineage visualization.
* **Custom Integration Needs**: The extensible REST API supports tailored integrations for enterprise workflows, ideal for organizations with complex ETL pipelines.

1. **Limitations**

* **Non-Hadoop Environments:** Less effective in modern data stacks without Hive or HBase, as integration with non-Hadoop systems (e.g., PostgreSQL, cloud-native platforms) requires custom development.
* **User Accessibility**: The UI’s complexity and slow performance for large lineage graphs hinder non-technical user adoption.
* **Rapid Deployment Needs:** Complex setup and configuration (e.g., HBase/Solr integration) make it less ideal for quick prototyping or agile environments.

**1.3.2 DataHub: Best Use Cases and Limitations**

1. **Best Use Cases**

* **Modern Data Stacks:** Ideal for organizations using cloud-native or hybrid data platforms (e.g., PostgreSQL, Snowflake, Redshift), as demonstrated in the POC with PostgreSQL integration and Airflow orchestration.
* **User-Friendly Metadata Management:** Suited for environments prioritizing accessibility for business users and data analysts, with an intuitive web UI for metadata exploration and search.
* **Balanced Governance and Discovery:** Effective for organizations needing both governance (e.g., PII/Sensitive tagging, ownership assignment) and self-service analytics, as shown with glossary term associations (e.g., Employee ID, Base Salary).

1. **Limitations**

* **Complex Lineage Needs:** Less robust than Atlas for detailed column-level lineage or deep Hadoop integration, limiting its use in governance-heavy Hadoop environments.
* **Small-Scale Deployments:** Resource requirements for Elasticsearch/MySQL make it overkill for small datasets or simple use cases.

**1.3.3 Amundsen : Prior Use Cases and Limitations**

1. **Prior Use Cases**

* **Self-Service Data Discovery**: Amundsen was previously used in environments where business users needed to explore datasets independently, leveraging its Flask/React frontend and Neo4j/Elasticsearch for intuitive search and visualization.
* **Lightweight Metadata Exploration:** Suited for organizations with simple metadata needs, where governance was secondary to discovery, supported by Databuilder for custom ingestion.
* **Modular Deployments:** Its microservice architecture (Frontend, Metadata, Search) was ideal for flexible setups in analytics-focused environments.

1. **Limitations**

* **Limited Documentation:** Documentation is less comprehensive, making troubleshooting harder.
* **Community-Driven Development:** Slower feature updates due to reliance on open-source contributions.
* **Scalability Concerns**: Neo4j and Elasticsearch may face performance issues with very large datasets.

# Chapter 2 Introduction

## 2.1 Background

The evolution of data architectures reflects the rapid growth in data complexity and the demand for agile, data-driven decision-making. In the 1970s to 1990s, organizations relied on monolithic, on-premises relational databases and early data warehouses. These systems used rigid schemas to manage structured data, often hosted on mainframes or client-server systems, with IT teams controlling access. However, their siloed nature, limited scalability, and inability to handle real-time analytics restricted decision-making to slow, batch-processed reports. Metadata was rudimentary, often confined to basic database schemas, and lacked systematic management, making data discovery and integration across departments challenging.

The early 2000s marked a shift with the rise of data warehousing and Extract, Transform, Load (ETL) processes, which centralized data from disparate sources into structured repositories for analytics. Technologies like Oracle, SQL Server, and later Hadoop enabled organizations to handle larger datasets and perform complex queries. This era introduced data lakes to accommodate unstructured and semi-structured data, driven by the big data boom. However, these systems struggled with data silos, governance issues, and the complexity of managing diverse data types. Metadata management emerged as critical, providing context (e.g., data lineage, schemas) to improve discoverability and trust, enabling faster and more reliable decision-making through better data accessibility.

Modern data architectures, such as cloud-native data lakes, lakehouses, and data meshes, prioritize flexibility, scalability, and real-time analytics. Technologies like Snowflake, Databricks, and AWS Redshift support distributed, decentralized data ecosystems, integrating structured and unstructured data across hybrid cloud environments. The data mesh paradigm, for instance, emphasizes domain-driven data ownership, treating data as a product. Metadata management has become indispensable in these architectures, acting as the backbone for data governance, discoverability, and compliance. Tools like Amundsen, DataHub, and Collibra provide robust metadata catalogs, enabling automated lineage tracking, data quality assurance, and regulatory compliance (e.g., GDPR). By ensuring data is findable, trustworthy, and accessible, metadata management empowers organizations to make informed, data-driven decisions in dynamic, complex environments.

## 2.2 Objectives

This report evaluates metadata management solutions using Apache Atlas and DataHub, noting Amundsen’s matured status and limited evaluation due to ingestion/UI issues. The objectives are to:

* Toassess Atlas and DataHub’s metadata ingestion, lineage, governance, and discovery via POCs with employee data, using Airflow, Spark, Hive (Atlas), and PostgreSQL (DataHub).
* To contrast Atlas and DataHub’s strengths, weaknesses, and performance to identify best use cases (e.g., governance for Atlas, user-friendly stacks for DataHub).
* To inform stakeholders on choosing tools for compliance, visibility, and analytics, with Atlas for Hadoop and DataHub for modern platforms.

# Chapter 3 Metadata Management Overview

## 3.1 Definition and Importance

Metadata is data that describes other data, providing context, structure, and meaning to facilitate its management, discovery, and use. It encompasses technical details (e.g., schema, data types, file formats), business context (e.g., ownership, glossary terms, classifications like PII), and operational information (e.g., lineage, processing timestamps). In the context of the POCs for Apache Atlas and DataHub, metadata includes schema definitions for employee data (e.g., CSV fields, Hive/PostgreSQL tables), lineage tracking (e.g., CSV → Spark → Hive), and governance tags (e.g., PII for sensitive fields like name and salary). By acting as a structured descriptor, metadata enables organizations to understand, organize, and leverage their data assets effectively.

Metadata plays a critical role in data governance by enabling organizations to enforce policies, ensure compliance, and maintain data quality. It provides visibility into data origins, transformations, and usage, which is essential for regulatory compliance (e.g., GDPR, HIPAA) and auditing. In the Atlas POC, metadata captured PII classifications and ownership details for employee data in Hive, supporting compliance through automated tagging and lineage visualization. Similarly, DataHub’s metadata management included glossary terms and ownership assignments, ensuring clear stewardship and auditable data flows. By documenting data attributes and relationships, metadata helps data stewards enforce standards, monitor sensitive data, and reduce risks associated with data misuse or breaches.

Metadata enhances data discovery by making datasets searchable and understandable, enabling self-service analytics for business users. In the DataHub POC, metadata like glossary terms (e.g., "Employee ID") and field descriptions facilitated user-friendly exploration via the web UI, while Atlas’s search capabilities allowed discovery of datasets and their lineage. For interoperability, metadata standardizes data descriptions across systems, ensuring seamless integration and data sharing. Atlas’s integration with Hive and DataHub’s with PostgreSQL demonstrated how metadata bridges heterogeneous platforms (e.g., Hadoop, modern data stacks), enabling consistent data access and collaboration across tools. Despite Amundsen’s ingestion/UI issues, its historical use for discovery relied on metadata to index and present data assets, underscoring its role in connecting users to relevant data.

## 3.2 Types of Metadata

### 3.2.1 Business Metadata

Business metadata provides critical context and meaning to data assets, enhancing their discoverability and usability for business stakeholders. In the Apache Atlas implementation, the system captures business metadata for an employee dataset processed through an ETL pipeline. It defines the dataset as containing employee information with fields such as id, name, age, department, and salary, where the department field includes values like "Engineering," "Marketing," "Sales," "Finance," "Human Resources," and "Executive." Business rules specify that the ETL pipeline, orchestrated by Apache Airflow, filters out employees under 18 years old (e.g., interns aged 16-17) using a Spark transformation, and salary data reflects varying compensation levels across departments. This metadata establishes the dataset’s business context and operational constraints.

In the DataHub implementation, business metadata enriches the "employees" dataset stored in PostgreSQL with comprehensive descriptions and custom properties. The system documents the dataset as "containing processed information about employees" via the datasetProperties aspect. It includes custom properties such as pipeline: "etl\_pipeline" to identify the data pipeline, team: "Data Team" to specify the responsible team, update\_frequency: "daily" to indicate refresh cadence, owner: "admin" for stewardship, data\_source: "ETL process" for origin, and created\_at: [timestamp] for creation tracking. Ownership is further detailed with a data owner (urn:li:corpuser:admin, type: DATAOWNER) and a technical developer (urn:li:corpuser:airflow, type: DEVELOPER). Additionally, DataHub associates glossary terms like "Employee ID" and "Base Salary" with dataset columns, ensuring business users understand the data’s purpose, ownership, and context.

### 3.2.2 Technical Metadata

Technical metadata describes the structure, format, and storage characteristics of data assets, enabling data engineers and analysts to understand and integrate data effectively. In the Apache Atlas implementation, the system captures technical metadata for the employee dataset transitioning from a CSV source to a Hive table (atlas\_test.transformed\_data). The source is a comma-separated CSV file, while the target Hive table has a schema comprising id (INT), name (STRING), department (STRING), and adjusted\_salary (DOUBLE). Data is stored as text files with comma delimiters in the Hive warehouse directory (/opt/hive/warehouse), with intermediate storage in Parquet format during Spark processing. Infrastructure details include the Hive Metastore running on thrift://hive-metastore:9083, Hive Server on port 10000, and Atlas on http://atlas:21000, providing a clear view of the data’s structure and storage environment.

In the DataHub implementation, technical metadata documents the structure of the employees table in PostgreSQL. The schema includes fields such as id (INTEGER, PRIMARY KEY), name (VARCHAR(255)), age (INTEGER), and salary (DECIMAL(10,2)), with field-level metadata specifying descriptions (e.g., "Employee ID - Primary Key"), native data types, and mappings to DataHub’s type system. The system also records the data platform (postgres), environment (PROD), and schema/namespace (etl\_pipeline). Additionally, the DDL representation is captured as CREATE TABLE employees (id INTEGER PRIMARY KEY, name VARCHAR(255), age INTEGER, salary DECIMAL(10,2)). This technical metadata ensures engineers understand the dataset’s structure, facilitating integration with analytical tools and downstream systems.

### 3.2.3 Operational Metadata

Operational metadata captures details about data processing, lineage, and usage, enabling teams to track data flows and ensure pipeline reliability. In the Apache Atlas implementation, the system records operational metadata for an ETL pipeline orchestrated by Apache Airflow. The pipeline extracts a CSV file from a GitHub repository, transforms it using Spark (e.g., filtering employees under 18), and loads it into a Hive table. The Atlas Hive hook (**org.apache.atlas.hive.hook.HiveHook,** configured with **atlas\_hook\_hive\_synchronous=true)** automatically captures lineage between the source CSV and the target Hive table, viewable in the Atlas UI. Additional metadata includes processing timestamps, verification steps to confirm metadata registration, and logs from Hive operations, all accessible via the Atlas UI. Integration occurs over a Docker network (etl\_network), with the Atlas cluster named "primary," ensuring robust tracking of data flows and operations.

In the DataHub implementation, operational metadata documents the ETL pipeline’s execution, orchestrated by Airflow with tasks to download a CSV, transform data using pandas (with Spark as a fallback), load into PostgreSQL, verify completeness, and publish metadata to DataHub. The system captures lineage tracing the flow from the CSV source to transformed PostgreSQL tables and analytical views, along with processing timestamps, actor information (urn:li:corpuser:airflow), and versioning for schema changes. Integration spans Airflow for orchestration, PostgreSQL for storage, and DataHub components (MySQL, Elasticsearch, Neo4j, Kafka) for metadata management. Health monitoring includes connection validation with fallback mechanisms, ensuring system reliability. This operational metadata provides transparency into data processing, supports troubleshooting, and enhances pipeline observability for technical teams.

# Chapter 4 Tools for Metadata Management

## 4.1 Installation and Configuration

### 4.1.1 Tool 1: Apache Atlas

1. **Prerequisites**

Before installing Apache Atlas, ensure the following prerequisites are met:

* Hardware Requirements:
  + - RAM: Minimum 16 GB.
    - CPU: At least 4 cores.
    - Disk Space: 100 GB free.
* Software Requirements:
  + For Docker Installation:
    - Docker (latest version recommended).
    - Docker Compose (latest version recommended).
  + Internet connection for downloading dependencies.
* User Access: Administrative or sudo privileges on the system.
* Network: Ensure port 21000 is available for the Atlas web UI.

1. **Installation Steps**
2. **Docker-Based Installation**

This method uses Docker to simplify setup and ensure consistency.

**Step 1: Create a Directory for Atlas Data:**

mkdir -p ~/atlas-data

chmod -R 777 ~/atlas-data

**Step 2: Create Docker Compose File**

* Create a file in the current directory

nano <docker\_compose\_file>

* Copy and paste the following content

Yaml

version: '3'

services:

  atlas:

    image: sburn/apache-atlas:2.3.0

    container\_name: atlas

    ports:

      - "21000:21000"

    environment:

      - ATLAS\_ENABLE\_TLS=false

      - ATLAS\_SERVER\_HEAP=-Xms1024m -Xmx1024m

    volumes:

      - ~/atlas-data:/opt/apache-atlas/data

    networks:

      - atlas\_network

    healthcheck:

      test: ["CMD", "curl", "--fail", "http://localhost:21000/api/atlas/admin/version"]

      interval: 30s

      timeout: 10s

      retries: 5

      start\_period: 60s

networks:

  atlas\_network:

    driver: bridge

**Step 3: Start Apache Atlas:**

* Run the Docker Compose file to start Atlas:

docker-compose -f docker-compose-atlas.yml up -d

* Wait 1-2 minutes for Atlas to initialize

**Step 4: Verify Installation:**

* Check if the Atlas container is running

docker ps -a | grep atlas

* View container logs for errors

docker logs atlas

* Confirm port 21000 is active

netstat -tuln | grep 21000

* Test the health check endpoint

curl http://localhost:21000/api/atlas/admin/version

* Expect a JSON response with the Atlas version (e.g., 2.3.0).
* Access the Atlas web UI
* Open a browser and navigate to <http://localhost:21000>.
* Log in with default credentials: username admin, password admin.

### 4.1.2 Tool 2: DataHub

1. **Prerequisites**

Before installing DataHub, Airflow, and PostgreSQL, ensure the following prerequisites are met:

* **Operating System**: Any system supporting Docker (e.g., Linux, macOS, Windows with WSL2).
* **Hardware Requirements**:
* **RAM**: Minimum 8 GB.
* **Disk Space**: At least 20 GB free.
* **Software Requirements**:
* Docker (latest version recommended).
* Docker Compose (latest version recommended).
* Python 3.8 or higher (for Airflow DAGs and scripting).
* **Network**: Ensure ports 8080 (Airflow), 8088 (DataHub GMS), 5433 (PostgreSQL), and 9002 (DataHub frontend) are available.
* **User Access**: Administrative or sudo privileges for Docker and file permissions.
* **Internet Connection**: Required for downloading Docker images and dependencies.

1. **Installation Steps**
2. **Docker-based Installation**

**Step 1: Set Up Workspace**

* Create a project directory

mkdir -p metadata\_ingestion

cd metadata\_ingestion

* Create subdirectories for Airflow

mkdir -p dags logs plugins

chmod -R 777 dags logs plugins

**Step 2: Create Docker Compose Configuration**

* Create a file named docker-compose.yml:

nano docker-compose.yml

* Copy and paste the following configuration, which includes PostgreSQL, DataHub core services (GMS, frontend, MySQL, Elasticsearch, Kafka), and Airflow:

yaml

networks:

  datahub-network:

    driver: bridge

volumes:

  postgres\_data:

  datahub\_mysql\_data:

  elasticsearch\_data:

services:

  postgres:

    image: postgres:13

    environment:

      POSTGRES\_USER: airflow

      POSTGRES\_PASSWORD: airflow

      POSTGRES\_DB: airflow

    ports:

      - "5433:5432"

    networks:

      - datahub-network

    healthcheck:

      test: ["CMD-SHELL", "pg\_isready -U airflow"]

      interval: 10s

      timeout: 5s

      retries: 5

    volumes:

      - postgres\_data:/var/lib/postgresql/data

  datahub-mysql:

    image: mysql:8.0

    environment:

      MYSQL\_ROOT\_PASSWORD: datahub

      MYSQL\_DATABASE: datahub

      MYSQL\_USER: datahub

      MYSQL\_PASSWORD: datahub

    networks:

      - datahub-network

    volumes:

      - datahub\_mysql\_data:/var/lib/mysql

  elasticsearch:

    image: elasticsearch:7.16.3

    environment:

      - discovery.type=single-node

      - ES\_JAVA\_OPTS=-Xms1g -Xmx1g

    networks:

      - datahub-network

    volumes:

      - elasticsearch\_data:/usr/share/elasticsearch/data

  datahub-gms:

    image: linkedin/datahub-gms:v0.9.0

    depends\_on:

      - datahub-mysql

      - elasticsearch

    environment:

      - EBEAN\_DATASOURCE\_USERNAME=datahub

      - EBEAN\_DATASOURCE\_PASSWORD=datahub

      - EBEAN\_DATASOURCE\_HOST=datahub-mysql

      - ELASTICSEARCH\_HOST=elasticsearch

    ports:

      - "8088:8080"

    networks:

      - datahub-network

  datahub-frontend:

    image: linkedin/datahub-frontend-react:v0.9.0

    depends\_on:

      - datahub-gms

    environment:

      - DATAHUB\_GMS\_HOST=datahub-gms

      - DATAHUB\_GMS\_PORT=8080

    ports:

      - "9002:9002"

    networks:

      - datahub-network

  kafka:

    image: confluentinc/cp-kafka:6.2.0

    environment:

      - KAFKA\_BROKER\_ID=1

      - KAFKA\_ZOOKEEPER\_CONNECT=zookeeper:2181

      - KAFKA\_ADVERTISED\_LISTENERS=PLAINTEXT://kafka:9092

      - KAFKA\_OFFSETS\_TOPIC\_REPLICATION\_FACTOR=1

    networks:

      - datahub-network

  zookeeper:

    image: confluentinc/cp-zookeeper:6.2.0

    environment:

      - ZOOKEEPER\_CLIENT\_PORT=2181

      - ZOOKEEPER\_TICK\_TIME=2000

    networks:

      - datahub-network

  airflow-init:

    image: apache/airflow:2.7.3

    command: bash -c "airflow db init && airflow users create --username admin --password admin --firstname Admin --lastname User --role Admin --email admin@example.com"

    environment:

      - AIRFLOW\_\_CORE\_\_SQL\_ALCHEMY\_CONN=postgresql+psycopg2://airflow:airflow@postgres:5432/airflow

    depends\_on:

      - postgres

    networks:

      - datahub-network

  airflow-webserver:

    image: apache/airflow:2.7.3

    environment:

      - AIRFLOW\_\_CORE\_\_EXECUTOR=LocalExecutor

      - AIRFLOW\_\_CORE\_\_SQL\_ALCHEMY\_CONN=postgresql+psycopg2://airflow:airflow@postgres:5432/airflow

    volumes:

      - ./dags:/opt/airflow/dags

      - ./logs:/opt/airflow/logs

      - ./plugins:/opt/airflow/plugins

    ports:

      - "8080:8080"

    depends\_on:

      - postgres

      - airflow-init

    networks:

      - datahub-network

**Step 3: Start Services**

* Launch all services

docker-compose up -d

* Wait 2-3 minutes for initialization of DataHub, Airflow, and PostgreSQL.
* Verify container status

docker ps -a

* Ensure all services (e.g., datahub-gms, datahub-frontend, airflow-webserver, postgres) are running (Up).

**Step 4: Verify Installation**

1. **PostgreSQL**

* Check PostgreSQL health

docker exec -it <postgres-container-id> pg\_isready -U airflow

* Expect output: /var/run/postgresql:5432 – accepting connections.
* Verify port 5433

netstat -tuln | grep 5433

1. **DataHub**

* Access the DataHub web UI:
* Navigate to <http://localhost:9002>
* Login with default credentials: username **datahub** , password **datahub** .
* Verify GMS service:

curl http://localhost:8088/health

* Expect a JSON response indicating service health.
* Check logs for error

docker logs <datahub-gms-container-id>

docker logs <datahub-frontend-container-id>

1. **Airflow**

* Access the Airflow web UI:
* Navigate to <http://localhost:8080>
* Log in with credentials: username **admin**, password **admin**.
* Verify Airflow database initialization

docker logs <airflow-init-container-id>

* Look for successful **db init** and user creation messages.
* Check logs for webserver

docker logs <airflow-webserver-container-id>

1. **Network Integration**

* Confirm all services are on the datahub-network:

docker network inspect datahub-network

* Verify containers (e.g., **postgres, datahub-gms, airflow-webserver**) are listed.

## 4.2 Challenges in Tool Installation

### Challenges in Installation : Apache Atlas

The installation of Apache Atlas for the Proof of Concept (POC) presented several challenges, reflecting its complexity as a metadata management and governance platform integrated with Apache Airflow, Hive, and Spark. These challenges, encountered during the setup of Atlas 2.3.0 in a Dockerized environment, spanned container configuration, network setup, Hive integration, resource management, and metadata ingestion. The following outlines the primary issues faced, their causes, and the solutions implemented, providing insights to streamline future deployments.

1. **Container Configuration and Version Compatibility**: Initial attempts to run Atlas, Hive, Spark, and Airflow in a single container led to resource constraints and port conflicts, exacerbated by version mismatches between Atlas 2.3.0, HBase, and Solr. Incompatible Hive versions caused Atlas Hive hook failures, preventing metadata capture. This was resolved these by adopting a multi-container Docker Compose setup for better isolation and pinning specific component versions (e.g., HBase 2.4.9, Solr 8.11.2) in the **docker-compose-atlas.yml** file, ensuring compatibility and stable service operation.
2. **Network Configuration Issues**: Services struggled with hostname resolution, notably Hive’s inability to connect to Atlas at http://atlas:21000, due to misconfigured Docker networks. Port conflicts on defaults like 8080, 9083, and 21000 further disrupted access. The solution involved creating a custom Docker network (etl\_network) with consistent hostnames and aliases, and remapping conflicting ports (e.g., 21000 to 21001 on the host), enabling seamless inter-service communication.
3. **Hive-Atlas Integration Challenges**: The Atlas Hive hook failed to capture metadata due to incorrect hive-site.xml configurations, missing JAR files, and improper hive.exec.post.hooks settings. Authentication issues also arose, with default credentials failing between Hive and Atlas. These was addressed by adding the Atlas Hive hook JARs to Hive’s classpath, setting **org.apache.atlas.hive.hook.HiveHook** in hive-site.xml, and configuring authentication in atlas-application.properties with proper user permissions, ensuring metadata flow from Hive tables.
4. **Initialization and Metadata Ingestion**: Services starting before dependencies (e.g., Hive before Atlas) caused failures, and initial metadata population was absent due to hook misconfigurations. Slow ingestion and timeouts occurred with large datasets. Health checks and depends\_on conditions were added in Docker Compose, introduced startup delays, and manually triggered metadata syncs after verifying hook settings. Tuning HBase/Solr configurations and implementing batch processing resolved performance issues, enabling reliable metadata ingestion for the employee dataset.

These challenges underscore the need for meticulous planning in Atlas deployments. Starting with a minimal configuration, maintaining version control for Docker Compose files, and implementing health checks from the outset were critical lessons. Comprehensive logging and monitoring, as achieved by mapping log directories and setting up health dashboards, facilitated debugging. These insights ensure more efficient installations, particularly for governance-heavy environments integrating with Hadoop ecosystems.

### 4.2.2 Challenges in Installation : DataHub

The installation of DataHub for the Proof of Concept (POC) presented several challenges, reflecting its complexity as a metadata management platform integrated with Apache Airflow and PostgreSQL in a Dockerized environment. These challenges, encountered during the setup of DataHub v0.9.0, spanned service initialization, network configuration, dependency integration, and resource management. The following outlines the primary issues faced, their root causes, and the solutions implemented, providing insights to streamline future deployments in modern data architectures.

1. **Service Initialization and Dependencies**: The DataHub Generalized Metadata Service (GMS) frequently failed to start due to uninitialized dependencies (Neo4j, Elasticsearch, Kafka). MySQL connection failures also occurred due to incorrect JDBC URLs and authentication settings. This was addressed by adding explicit depends\_on conditions and health checks in the docker-compose.yml file for services like datahub-mysql and elasticsearch, setting a restart: on-failure policy, and configuring the MySQL jdbc connection.
2. **Frontend and Authentication Issues**: The DataHub frontend could not connect to the GMS due to incorrect host configurations, and login attempts with default credentials failed. This was resolved by setting accurate environment variables (DATAHUB\_GMS\_HOST: datahub-gms, DATAHUB\_GMS\_PORT: 8080) in the frontend service configuration and enabling authentication with explicit credentials (DATAHUB\_AUTH\_ENABLED: "true", DATAHUB\_ROOT\_USERNAME: admin, DATAHUB\_ROOT\_PASSWORD: admin), allowing seamless UI access at <http://localhost:9002>.
3. **Network Configuration and Service Discovery**: Services struggled to resolve each other’s hostnames, hindering communication between GMS, frontend, and dependencies. A consistent Docker network (etl\_net) with aliases for key services (e.g., datahub-gms, datahub-frontend) in the docker-compose.yml was implemented, enabling proper service discovery and eliminating hostname resolution errors across containers.

These challenges highlight the importance of precise dependency management and network configuration in DataHub deployments. Key lessons include starting with validated environment variables, implementing health checks early, and ensuring sufficient resource allocation. Maintaining detailed logs (mapped to host directories) and monitoring service health via Docker Compose health checks were critical for debugging. These insights facilitate smoother installations, particularly for user-friendly metadata management in modern data stacks integrating with PostgreSQL and Airflow.

# Chapter 5 Capabilities Comparison

## 5.1 Comparative Analysis Framework

The comparison of Apache Atlas and DataHub focuses on evaluating their capabilities as metadata management tools, with a particular emphasis on metadata ingestion and related functionalities within modern data architectures. The framework establishes a set of criteria to assess their suitability for enterprise use cases, drawing from their implementation experiences and observed features. The defined criteria are categorized into key areas to ensure a comprehensive evaluation:

1. **General Information**:
   1. **Tool Origin & Backing Community**: Assesses the origin and the strength of the supporting community to gauge long-term viability and support.
   2. **License Type**: Evaluates the licensing model (e.g., Apache 2.0) to determine openness and usage flexibility.
   3. **Active Development & Community Support**: Measures the frequency of updates and the quality of community engagement via forums or Slack.
2. **Core Metadata Features**:
   1. **Metadata Ingestion Support**: Evaluates support for batch, streaming, push/pull mechanisms to assess ingestion versatility.
   2. **Data Lineage Tracking**: Analyzes the granularity (column, table, job-level) of lineage tracking capabilities.
   3. **Metadata Search**: Assesses the effectiveness of full-text and faceted search for metadata discovery.
   4. **Metadata Versioning**: Checks the ability to track and manage metadata change history.
   5. **Schema Change Tracking**: Reviews automation and support for evolving metadata structures.
   6. **Custom Metadata Support**: Measures flexibility in adding tags, glossary terms, and annotations.
3. **Data Lineage & Impact Analysis**:
   1. **Lineage Visualization**: Evaluates the clarity and usability of lineage representation in the UI.
   2. **Cross-System Lineage Support**: Assesses support for lineage across diverse data systems.
   3. **Upstream/Downstream Impact Analysis**: Examines the ability to analyze data flow impacts.
   4. **Auto-Discovery of Lineage**: Reviews the automation level of lineage detection during ingestion.
4. **Search & Discovery**:
   1. **Search UX**: Evaluates user experience with natural language, autocomplete, and filters.
   2. **Faceted Search**: Assesses search refinement by tags, owners, and systems.
   3. **Relationship-Based Exploration**: Reviews the ability to navigate data relationships.
5. **Community and Ecosystem**:
   1. **Number of Contributors**: Gauges the size and activity of the contributor base.
   2. **Open-Source vs Enterprise Features**: Compares open-source offerings with enterprise add-ons.
   3. **Documentation Quality**: Assesses clarity and comprehensiveness of documentation.
   4. **Slack/Discord or Mailing List Support**: Evaluates the availability and responsiveness of support channels.

These criteria provide a structured and detailed basis for comparing Apache Atlas and DataHub, focusing on their metadata ingestion strengths, governance capabilities, and ecosystem

## 5.2 Comparative Matrix

1. **General Information**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Apache Atlas** | **DataHub** |
| Tool Origin & Backing Community | Originated from Apache Software Foundation; backed by a broad open-source community with contributions from Hortonworks (now Cloudera). | Originated from LinkedIn; maintained by Acryl Data with a growing open-source community. |
| License Type | Apache 2.0 License. | Apache 2.0 License. |
| Active Development & Community Support | Actively developed with consistent releases; strong community support via Apache mailing lists and forums. | Actively developed with frequent updates; community support via Slack and GitHub discussions. |

Table 5.I. 1 General Comparison Apache Atlas Vs DataHub

1. **Core Metadata Features**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Apache Atlas** | **DataHub** |
| Metadata Ingestion Support | Supports batch ingestion via Hive hooks and APIs; limited streaming ingestion capabilities. | Supports batch ingestion via Airflow DAGs and push/pull mechanisms; lacks robust streaming ingestion. |
| Data Lineage Tracking | Comprehensive lineage tracking at table and job levels; column-level lineage requires additional setup. | Table and job-level lineage tracking; column-level lineage support is incomplete. |
| Metadata Search | Full-text search with Solr integration; faceted search for classifications and entities. | Full-text search with Elasticsearch; faceted search by tags |
| Metadata Versioning | Limited versioning; tracks metadata changes but lacks explicit version history. | Supports metadata versioning with historical tracking of changes via audit logs. |
| Schema Change Tracking | Manual schema change tracking; no automated detection of schema evolution. | Manual schema updates required; no automated schema change detection. |
| Custom Metadata Support | Supports tags and glossary terms; limited custom annotations and attribute support. | Strong support for custom metadata (tags, glossary terms, annotations) via API and UI. |

Table 5.II. 1 Core Metadata Features Comparison Apache Atlas Vs DataHub

1. **Data Lineage and Impact Analysis**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Apache Atlas** | **DataHub** |
| Lineage Visualization | Robust lineage visualization in UI | Clear lineage visualization with graph-based UI for datasets and pipelines. |
| Cross-System Lineage Support | Strong cross-system lineage for Hive | Spark and Airflow; limited for unsupported sources. |
| Upstream/Downstream Impact Analysis | Provides upstream/downstream impact analysis. | Basic upstream/downstream analysis; lacks depth for complex dependencies. |
| Auto-Discovery of Lineage | Automatic lineage discovery via Hive hooks during ETL processes. | Limited auto-discovery; relies on manual configuration or Airflow metadata ingestion. |

Table 5.III. 1 Data lineage and Impact Analysis Comparison Apache Atlas Vs DataHub

1. **Search and discoverability**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Apache Atlas** | **DataHub** |
| Search UX | Basic search UX with full-text support; lacks natural language or autocomplete features. | Enhanced search UX with autocomplete filters and natural language query support. |
| Faceted Search | Faceted search by classifications, entities, and systems; less intuitive for non-technical users. | Faceted search by tags, owners, and systems; user-friendly for all audiences. |
| Relationship-Based Exploration | Supports exploration of relationships (datasets to pipelines); UI can be complex. | Strong relationship-based exploration (datasets to pipelines, dashboards); intuitive navigation. |

Table 5.IV. 1 Search and Discoverablilty Comparison Apache Atlas Vs DataHub

1. **Community and Ecosystem**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Apache Atlas** | **DataHub** |
| Number of Contributors | Large contributor base (~100+ active contributors) under Apache Foundation. | Smaller but growing contributor base (~50+ active contributors) via Acryl Data. |
| Open-Source vs Enterprise Features | Fully open-source; no enterprise tier but integrates with Cloudera’s enterprise tools. | Open-source with enterprise features (e.g., advanced governance) via Acryl Data. |
| Documentation Quality | Comprehensive but technical documentation; can be overwhelming for beginners. | Well-structured documentation with tutorials; more accessible for new users. |
| Slack/Discord or Mailing List Support | Strong support via Apache mailing lists and forums. | Active Slack community and GitHub discussions for support. |

Table 5.V. 1 Community and EcoSystem Comparison Apache Atlas Vs DataHub

## 5.3 Analysis of Strengths and Weaknesses

1. **Apache Atlas**

**Strengths:** Apache Atlas stands out in several areas critical for large-scale, governance-heavy metadata management. Its data lineage and impact analysis capabilities are robust, with automatic lineage discovery via Hive hooks, comprehensive table and job-level lineage tracking, and strong cross-system lineage support for Hive, Spark, and Airflow. This makes it ideal for ETL-centric environments where real-time lineage tracking is essential. The general information category highlights its strength with a large contributor base (~100+ active contributors) under the Apache Foundation, ensuring consistent development and strong mailing list support. Additionally, its collaboration features include robust glossary and taxonomy management, which are valuable for enterprise governance workflows.

**Weaknesses:** Apache Atlas falls short in areas requiring flexibility and modern usability. In core metadata features, it struggles with limited streaming ingestion support, manual schema change tracking, and restricted custom metadata support, which hampers adaptability. Its UI/UX and user experience are less intuitive, with a steep learning curve, no dark/light theme support, and limited accessibility, making it less suitable for business users. Search and discovery capabilities lag with a basic search UX lacking natural language or autocomplete features, and faceted search is less user-friendly for non-technical users. In collaboration features, Atlas lacks commenting and discussion threads, and its stewardship workflows are manual.

1. **DataHub**

**Strengths:** DataHub shines in areas focused on usability, extensibility, and modern deployment. Its UI/UX and user experience are a standout, with a highly intuitive interface, customizable dashboards, dark/light theme support, and accessibility features, making it accessible to both business and technical users. In search and discovery, DataHub offers an enhanced search UX with autocomplete, filters, and natural language query support, alongside intuitive relationship-based exploration of datasets, pipelines, and dashboards. Collaboration features are strong, with support for commenting, discussion threads, automated ownership assignment, and advanced alerts/notifications, fostering better team collaboration. In core metadata features, DataHub provides strong support for custom metadata (tags, glossary terms, annotations) and metadata versioning with historical tracking.

**Weaknesses:** DataHub has notable gaps in governance and lineage depth. In data lineage and impact analysis, it lacks robust column-level lineage, auto-discovery is limited, and upstream/downstream impact analysis is basic, making it less suitable for complex dependency tracking. Data governance and security are weaker, with less granular RBAC, basic policy management, and limited GDPR/HIPAA support (compliance features are enterprise-only or manual). In core metadata features, DataHub struggles with a lack of streaming ingestion support, manual schema change tracking, and incomplete automation for lineage and governance tasks. Finally, its community and ecosystem is smaller (~50+ contributors) compared to Atlas, and some advanced features (e.g., governance) are locked behind Acryl Data’s enterprise tier, potentially limiting open-source users.

# Chapter 6 Implementations

## Workflow Overview

### 6.1.1 DATAHUB Workflow Overview

The csv\_etl\_pipeline DAG orchestrates a sequence of tasks to process employee data and integrate its metadata into DataHub. The workflow is designed to run once (@once schedule) and is manually triggered, with a start date of January 1, 2024. It includes retry logic (2 retries, 2-minute delay) to handle transient failures. The workflow consists of seven tasks, each with a specific role in the ETL process and metadata emission, executed in a linear dependency chain as depicted in the following diagram:

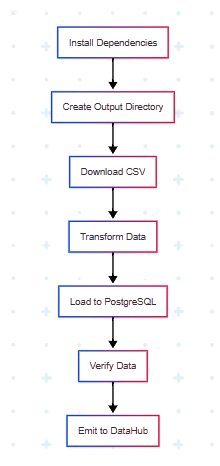


Figure 6.1. 1 DAG for metadata ingestion in DataHub

### 6.1.2 Apache Atlas

The atlas\_integration\_final.py DAG automates the process of integrating Apache Atlas with Hive to capture metadata from a simulated ETL pipeline. The DAG is configured to run manually (schedule\_interval=None) with default arguments specifying the owner, start date, and retry settings. It consists of five tasks executed in a linear dependency chain, ensuring each step completes successfully before the next begins. The workflow verifies Atlas connectivity, simulates an ETL process, ingests metadata into Atlas, verifies the ingested metadata, and simulates the creation of a metadata dashboard. The workflow diagram is presented below:

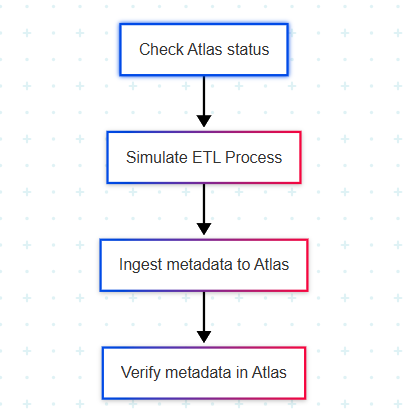


Figure 6.1. 2 DAG for metadata ingestion in Apache Atlas

## 6.2 Task Breakdown

### 6.2.1 DATAHUB Metadata Ingestion Tasks Breakdown and Implementation guide

The DAG automates an ETL process with metadata ingestion into DataHub, a metadata management platform. It consists of six tasks executed sequentially: installing dependencies, creating an output directory, downloading a CSV file, transforming the data, loading it into PostgreSQL, verifying the data, and emitting metadata to DataHub. The DAG is configured with default arguments, including an owner (airflow), start date (January 1, 2024), retry settings (2 retries, 2-minute delay), and a one-time schedule (@once).

The pipeline performs the following:

* **Installs dependencies** to ensure required Python packages are available.
* **Creates an output directory** for storing temporary files.
* **Downloads a CSV file** from a GitHub URL containing employee data (id, age, name, salary).
* **Transforms the data** by cleaning and casting columns.
* **Loads the data** into a PostgreSQL table (employees) with columns id, age, name, salary.
* **Verifies the data** to ensure successful loading and data integrity.
* **Emits metadata** to DataHub, including dataset properties, schema metadata id, age, name, salary), and ownership information.

1. **Install Dependencies (**install\_dependencies**)**

* **Type**: BashOperator
* **Purpose**: Installs required Python packages (pandas, sqlalchemy, psycopg2-binary, requests) to support subsequent tasks.
* **Execution**: Runs the command pip install pandas sqlalchemy psycopg2-binary requests in the Airflow environment.
* **Role in Workflow**: Ensures the Airflow environment has the necessary libraries for data processing and HTTP requests, laying the foundation for all subsequent tasks.
* **Dependencies**: None (starting task).
* **Significance for Metadata Ingestion**: Provides the requests library, which is critical for the emit\_to\_datahub task to communicate with DataHub’s metadata service (e.g., via REST API).

```bash

pip install pandas sqlalchemy psycopg2-binary requests && echo "Dependencies installed"

```

1. **Create Output Directory (**create\_output\_directory**)**

* **Type**: PythonOperator
* **Function**: create\_output\_dir()
* **Purpose**: Creates the /tmp/spark\_output directory to store intermediate files (e.g., downloaded and transformed CSVs).
* **Execution**: Executes a Python function to create the directory, ensuring a clean workspace for file operations.
* **Dependencies**: Depends on install\_dependencies to ensure the environment is set up.
* **Role in Workflow**: Provides a designated location for storing the CSV files generated during the ETL process, ensuring subsequent tasks can read/write files without errors.
* **Significance for Metadata Ingestion**: Ensures the file-based data assets (e.g., /tmp/employee\_data.csv, /tmp/transformed\_data.csv) are properly managed, which is critical for tracking dataset metadata in DataHub.

def create\_output\_dir():

    output\_dir = '/tmp/spark\_output'

    if not os.path.exists(output\_dir):

        os.makedirs(output\_dir)

    return output\_dir

1. **Download CSV (**download\_csv**)**

* **Type**: PythonOperator
* **Function**: download\_csv(\*\*kwargs)
* **Purpose**: Downloads employee data from a GitHub raw content URL and saves it as /tmp/employee\_data.csv.
* **Execution**: Uses the requests library to fetch the CSV file and writes it to the specified path.
* **Dependencies**: Depends on create\_output\_directory to ensure the output directory exists.
* **Role in Workflow**: Extracts the raw data from an external source, initiating the ETL pipeline by providing the input dataset.
* **Significance for Metadata Ingestion**: The downloaded CSV represents the source dataset, whose metadata (e.g., dataset properties, schema) will be ingested into DataHub in the final task.

def download\_csv(\*\*kwargs):

    url = "https://raw.githubusercontent.com/Subashkhanal2580/metadata\_test/main/employee\_data.csv"

    local\_path = "/tmp/employee\_data.csv"

    response = requests.get(url)

    with open(local\_path, 'wb') as f:

        f.write(response.content)

    print(f"Downloaded CSV from {url} to {local\_path}")

    df = pd.read\_csv(local\_path)

    print(f"CSV has {len(df)} rows and {len(df.columns)} columns")

    print(f"Sample data:\n{df.head(3)}")

    return local\_path

1. **Transform Data (**transform\_data**)**

* **Type**: PythonOperator
* **Function**: transform\_data(\*\*kwargs)
* **Purpose**: Processes the raw CSV (/tmp/employee\_data.csv) by filling missing values, converting data types, and performing validation checks, then saves the result as /tmp/transformed\_data.csv.
* **Execution**: Uses pandas for data manipulation, applying transformations such as handling NULLs and ensuring data consistency.
* **Dependencies**: Depends on download\_csv to ensure the input CSV exists.
* **Role in Workflow**: Transforms raw data into a clean, consistent format suitable for loading into the PostgreSQL database.
* **Significance for Metadata Ingestion**: The transformation step defines the schema and properties of the processed dataset, which are captured as metadata (e.g., column types, dataset description) for ingestion into DataHub.

def transform\_data(\*\*kwargs):

    input\_file = kwargs['ti'].xcom\_pull(task\_ids='download\_csv')

    output\_file = "/tmp/transformed\_data.csv"

    df = pd.read\_csv(input\_file)

    print(f"Original data shape: {df.shape}")

    df['age'] = df['age'].fillna(0).astype(int)

    df['name'] = df['name'].fillna('Unknown')

    df['salary'] = df['salary'].fillna(0).astype(float)

    df = df[df['id'].notna()]

    df['id'] = df['id'].astype(int)

    print(f"Transformed data shape: {df.shape}")

    print(f"Sample transformed data:\n{df.head(3)}")

    df.to\_csv(output\_file, index=False)

    print(f"Transformed data saved to {output\_file}")

    return output\_file

1. **Load to PostgreSQL (**load\_to\_postgres**)**

* **Type**: PythonOperator
* **Function**: load\_to\_postgres(\*\*kwargs)
* **Purpose**: Loads the transformed data (/tmp/transformed\_data.csv) into a PostgreSQL database table named employees at postgresql://airflow:airflow@postgres:5432/etl\_pipeline.
* **Execution**: Uses sqlalchemy and psycopg2-binary to connect to the PostgreSQL database and insert the data.
* **Dependencies**: Depends on transform\_data to ensure the transformed CSV is available.
* **Role in Workflow**: Loads the cleaned data into the target database, completing the ETL process’s loading phase.
* **Significance for Metadata Ingestion**: The employees table in PostgreSQL becomes a key dataset tracked in DataHub, with its schema, ownership, and properties emitted as metadata.

def load\_to\_postgres(\*\*kwargs):

    transformed\_file = kwargs['ti'].xcom\_pull(task\_ids='transform\_data')

    df = pd.read\_csv(transformed\_file)

    print(f"Loading {len(df)} rows to PostgreSQL")

    conn\_str = "postgresql+psycopg2://airflow:airflow@postgres:5432/etl\_pipeline"

    engine = create\_engine(conn\_str)

    try:

        with engine.connect() as connection:

            connection.execute("""

            CREATE TABLE IF NOT EXISTS employees (

                id INTEGER PRIMARY KEY,

                age INTEGER,

                name VARCHAR(255),

                salary DECIMAL(10, 2)

            )

            """)

            connection.execute("TRUNCATE TABLE employees")

        df.to\_sql('employees', engine, if\_exists='append', index=False)

        print("Data successfully loaded to PostgreSQL")

        with engine.connect() as connection:

            result = connection.execute("SELECT COUNT(\*) FROM employees")

            count = result.fetchone()[0]

            print(f"Verified {count} rows in the employees table")

        return count

    except Exception as e:

        print(f"Error loading data to PostgreSQL: {str(e)}")

        raise

1. **Verify Data (**verify\_data**)**

* **Type**: PythonOperator
* **Function**: verify\_data(\*\*kwargs)
* **Purpose**: Validates the loaded data in the employees table by checking row counts, detecting NULL values, and sampling data for quality assurance.
* **Execution**: Queries the PostgreSQL database to perform validation checks, ensuring data integrity before metadata emission.
* **Dependencies**: Depends on load\_to\_postgres to ensure the data is loaded into the database.
* **Role in Workflow**: Confirms that the data loaded into PostgreSQL is complete and correct, preventing erroneous metadata from being emitted to DataHub.
* **Significance for Metadata Ingestion**: Ensures the dataset’s metadata reflects accurate and validated data, enhancing trust in DataHub’s metadata records.

def verify\_data(\*\*kwargs):

    conn\_str = "postgresql+psycopg2://airflow:airflow@postgres:5432/etl\_pipeline"

    engine = create\_engine(conn\_str)

    try:

        with engine.connect() as connection:

            result = connection.execute("SELECT COUNT(\*) FROM employees")

            count = result.fetchone()[0]

            if count == 0:

                raise ValueError("No data loaded into the employees table")

            print(f"Found {count} rows in the employees table")

            result = connection.execute("SELECT COUNT(\*) FROM employees WHERE id IS NULL")

            null\_count = result.fetchone()[0]

            if null\_count > 0:

                print(f"WARNING: Found {null\_count} rows with NULL IDs")

            result = connection.execute("SELECT \* FROM employees LIMIT 5")

            rows = result.fetchall()

            print("Sample data from employees table:")

            for row in rows:

                print(row)

            return count

    except Exception as e:

        print(f"Error verifying data: {str(e)}")

        raise

1. **Emit to DataHub (**emit\_to\_datahub**)**

* **Type**: PythonOperator
* **Function**: emit\_to\_datahub(\*\*kwargs)
* **Purpose**: Emits metadata about the employees dataset to DataHub, including dataset properties, schema definition, and ownership information.
* **Execution**: Uses the DataHub Python SDK or REST API to send metadata to the DataHub metadata service (e.g., datahub-gms on port 8080).
* **Dependencies**: Depends on verify\_data to ensure the dataset is valid before emitting metadata.
* **Role in Workflow**: Integrates the ETL pipeline with DataHub by registering the dataset’s metadata, enabling governance, discovery, and lineage tracking.
* **Significance for Metadata Ingestion**: This task is the culmination of the workflow, where the metadata (e.g., dataset URN, schema, owners) is ingested into DataHub, making the dataset discoverable and governable.

def emit\_to\_datahub(\*\*kwargs):

    """

    Emit metadata to DataHub using direct REST API call to the correct endpoints.

    """

    dataset\_urn = "urn:li:dataset:(urn:li:dataPlatform:postgres,etl\_pipeline.employees,PROD)"

    gms\_endpoint = "http://datahub-gms:8080"

    print(f"Using dataset URN: {dataset\_urn}")

    # dataset description properties

    print(f"Sending dataset properties to {gms\_endpoint}")

    dataset\_properties = {

        "description": "Employee dataset that contains processed information about employees",

        "customProperties": {

            "pipeline": "etl\_pipeline",

            "team": "Data Team",

            "update\_frequency": "daily"

        }

    }

    # request payloads for each aspect

    aspect\_requests = [

        # Dataset properties

        {

            "proposal": {

                "entityType": "dataset",

                "entityUrn": dataset\_urn,

                "changeType": "UPSERT",

                "aspectName": "datasetProperties",

                "aspect": {

                    "value": json.dumps(dataset\_properties),

                    "contentType": "application/json"

                }

            }

        },

        # Schema metadata

        {

            "proposal": {

                "entityType": "dataset",

                "entityUrn": dataset\_urn,

                "changeType": "UPSERT",

                "aspectName": "schemaMetadata",

                "aspect": {

                    "value": json.dumps({

                        "schemaName": "employees\_schema",

                        "platform": "urn:li:dataPlatform:postgres",

                        "version": 1,

                        "created": {

                            "time": int(time.time() \* 1000),

                            "actor": "urn:li:corpuser:etl"

                        },

                        "lastModified": {

                            "time": int(time.time() \* 1000),

                            "actor": "urn:li:corpuser:etl"

                        },

                        "hash": "",

                        "platformSchema": {

                            "com.linkedin.schema.MySqlDDL": {

                                "tableSchema": "CREATE TABLE employees (id INT, name VARCHAR(100), department VARCHAR(100));"

                            }

                        },

                        "fields": [

                            {

                                "fieldPath": "id",

                                "description": "Employee ID",

                                "type": {"type": {"com.linkedin.schema.NumberType": {}}},

                                "nativeDataType": "INTEGER"

                            },

                            {

                                "fieldPath": "name",

                                "description": "Employee Name",

                                "type": {"type": {"com.linkedin.schema.StringType": {}}},

                                "nativeDataType": "VARCHAR"

                            },

                            {

                                "fieldPath": "department",

                                "description": "Employee Department",

                                "type": {"type": {"com.linkedin.schema.StringType": {}}},

                                "nativeDataType": "VARCHAR"

                            }

                        ]

                    }),

                    "contentType": "application/json"

                }

            }

        },

        # Ownership

        {

            "proposal": {

                "entityType": "dataset",

                "entityUrn": dataset\_urn,

                "changeType": "UPSERT",

                "aspectName": "ownership",

                "aspect": {

                    "value": json.dumps({

                        "owners": [

                            {

                                "owner": "urn:li:corpuser:admin",

                                "type": "DATAOWNER"

                            }

                        ],

                        "lastModified": {

                            "time": int(time.time() \* 1000),

                            "actor": "urn:li:corpuser:etl"

                        }

                    }),

                    "contentType": "application/json"

                }

            }

        }

    ]

    success\_count = 0

    for request\_data in aspect\_requests:

        aspect\_name = request\_data["proposal"]["aspectName"]

        print(f"Sending {aspect\_name} to DataHub...")

        try:

            # ingestProposal endpoint

            response = requests.post(

                f"{gms\_endpoint}/aspects?action=ingestProposal",

                headers={

                    "Content-Type": "application/json",

                    "X-RestLi-Protocol-Version": "2.0.0"

                },

                json=request\_data

            )

            if response.status\_code == 200:

                print(f"Successfully emitted {aspect\_name}: {response.json()}")

                success\_count += 1

            else:

                print(f"Failed to emit {aspect\_name}: {response.status\_code} - {response.text}")

        except Exception as e:

            print(f"Error during {aspect\_name} emission: {str(e)}")

    if success\_count != len(aspect\_requests):

        raise Exception(f"Failed to emit all metadata to DataHub. Only {success\_count}/{len(aspect\_requests)} aspects were successful.")

    print(f"Successfully emitted {success\_count} aspects to DataHub.")

    return "Metadata ingestion completed"

#### **Key Considerations for Datahub metadata ingestion setup**

1. **Environment Setup**: The install\_dependencies task ensures all required libraries are available, particularly requests for DataHub API calls and psycopg2-binary for PostgreSQL connectivity.
2. **Data Quality**: The transform\_data and verify\_data tasks ensure the dataset is clean and validated, which is critical for accurate metadata ingestion.
3. **Network Configuration**: The workflow assumes that the Airflow environment, PostgreSQL, and DataHub services (e.g., datahub-gms) are running in a Docker network (e.g., datahub\_network) with proper hostname resolution (e.g., postgres, datahub-gms).
4. **Metadata Completeness**: The emit\_to\_datahub task should include comprehensive metadata, such as:

* **Dataset Properties**: Name, description, and source (e.g., GitHub URL).
* **Schema Definition**: Column names, data types, and constraints from the employees table.
* **Ownership Information**: Owners or responsible teams, enhancing governance in DataHub.

#### **Integrations for Datahub metadata ingestion process**

1. **Connection to PostgreSQL database**: The DAG establishes a connection to a PostgreSQL database using SQLAlchemy with the connectionpostgresql+psycopg2://airflow:airflow@postgres:5432/etl\_pipeline. This ensures secure and efficient communication with the database for data loading and verification tasks.

def load\_to\_postgres(\*\*kwargs):

    transformed\_file = kwargs['ti'].xcom\_pull(task\_ids='transform\_data')

    df = pd.read\_csv(transformed\_file)

    conn\_str = "postgresql+psycopg2://airflow:airflow@postgres:5432/etl\_pipeline"

    engine = create\_engine(conn\_str)

    try:

        with engine.connect() as connection:

            connection.execute("""

            CREATE TABLE IF NOT EXISTS employees (

                id INTEGER PRIMARY KEY,

                age INTEGER,

                name VARCHAR(255),

                salary DECIMAL(10, 2)

            )

            """)

            connection.execute("TRUNCATE TABLE employees")

        df.to\_sql('employees', engine, if\_exists='append', index=False)

1. **DataHub Metadata Integration:** : The emit\_to\_datahub task interacts with DataHub’s General Metadata Service (GMS) via its REST API, typically at http://datahub-gms:8080. This enables programmatic ingestion of metadata about the employees dataset.

def emit\_to\_datahub(\*\*kwargs):

    dataset\_urn = "urn:li:dataset:(urn:li:dataPlatform:postgres,etl\_pipeline.employees,PROD)"

    gms\_endpoint = "http://datahub-gms:8080"

    dataset\_properties = {

        "description": "Employee dataset that contains processed information about employees",

        "customProperties": {

            "pipeline": "etl\_pipeline",

            "team": "Data Team",

            "update\_frequency": "daily"

        }

    }

    response = requests.post(

        f"{gms\_endpoint}/aspects?action=ingestProposal",

        headers={

            "Content-Type": "application/json",

            "X-RestLi-Protocol-Version": "2.0.0"

        },

        json=request\_data

    )

1. **GitHub Data Source Integration:** The download\_csv task uses the requests library to fetch employee data from a GitHub raw content URL, saving it as /tmp/employee\_data.csv.

def download\_csv(\*\*kwargs):

    url = "https://raw.githubusercontent.com/Subashkhanal2580/metadata\_test/main/employee\_data.csv"

    local\_path = "/tmp/employee\_data.csv"

    response = requests.get(url)

    with open(local\_path, 'wb') as f:

        f.write(response.content)

    print(f"Downloaded CSV from {url} to {local\_path}")

    df = pd.read\_csv(local\_path)

    return local\_path

1. **XCom Communication**: Uses Airflow’s XCom system to pass critical data between tasks, such as file paths (e.g., from download\_csv to transform\_data) and record counts for validation (e.g., from load\_to\_postgres to verify\_data).

# Passing file path from download\_csv to transform\_data

def transform\_data(\*\*kwargs):

    input\_file = kwargs['ti'].xcom\_pull(task\_ids='download\_csv')

    output\_file = "/tmp/transformed\_data.csv"

    return output\_file

# Passing record count from load\_to\_postgres to verify\_data

def load\_to\_postgres(\*\*kwargs):

    with engine.connect() as connection:

        result = connection.execute("SELECT COUNT(\*) FROM employees")

        count = result.fetchone()[0]

    return count

def verify\_data(\*\*kwargs):

    conn\_str = "postgresql+psycopg2://airflow:airflow@postgres:5432/etl\_pipeline"

    engine = create\_engine(conn\_str)

    try:

        with engine.connect() as connection:

            result = connection.execute("SELECT COUNT(\*) FROM employees")

            count = result.fetchone()[0]

### 6.2.2 Apache Atlas Metadata Ingestion Tasks Breakdown and Implementation guide

The DAG integrates with Apache Atlas to capture and manage metadata for ETL processes involving Hive tables. It uses PythonOperator tasks to execute Python functions and interacts with HiveServer2 (via Beeline) and Apache Atlas (via HTTP API). The DAG is configured with default arguments, including an owner (airflow), start date (May 19, 2025), retry settings (2 retries, 60-second delay), and a manual schedule (schedule\_interval=None).

The pipeline performs the following:

* **Checks Atlas connectivity** to ensure the service is operational.
* **Creates a sample Hive table** (atlas\_test.employees) to provide initial data.
* **Simulates an ETL process** by creating and transforming additional Hive tables (source\_data, transformed\_data, analytics\_data).
* **Ingests metadata** into Atlas, capturing lineage for the ETL process.
* **Verifies metadata** presence and lineage in Atlas.

1. **Check Atlas Status (**check\_atlas\_status**)**

* **Type**: PythonOperator
* **Function**: check\_atlas\_status()
* **Purpose**: Verifies that the Apache Atlas server is running and accessible, ensuring the metadata ingestion environment is ready.
* **Execution**: Creates an instance of the AtlasMetadataIngestion class and makes an HTTP request to the Atlas admin version endpoint (http://atlas:21000/api/atlas/admin/version). Returns True if the connection is successful, False otherwise, with clear console output indicating the connection status.
* **Dependencies**: None (starting task).
* **Role in Workflow**: Acts as a gatekeeper, ensuring Atlas is operational before proceeding with ETL and metadata ingestion tasks. If this task fails, the DAG fails fast, preventing unnecessary execution of subsequent tasks.
* **Significance for Metadata Ingestion**: Confirms that the Atlas server is available for metadata ingestion, avoiding failures in later tasks that rely on the Atlas API.

def check\_atlas\_status():

    """Check if Atlas is up and running."""

    atlas\_url = "http://atlas:21000"

    try:

        # Atlas client instance

        atlas\_client = AtlasMetadataIngestion(atlas\_url)

        # connection verification

        if atlas\_client.check\_connection():

            print("Atlas connection successful")

            return True

        else:

            print("Could not connect to Atlas")

            return False

    except Exception as e:

        print(f"Error checking Atlas status: {str(e)}")

        raise

1. **Simulate ETL Process (**simulate\_etl\_process**)**

* **Type**: PythonOperator
* **Function**: create\_hive\_table()
* **Purpose**: Simulates an ETL process by creating a sample database (atlas\_test) and an employees table in Hive, populating it with test data and performing simulated transformations.
* **Execution**: Executes Hive SQL commands via Beeline (Hive’s JDBC interface) to create the database and table, insert sample data, and simulate data transformations. Includes detailed logging for success and failure cases.
* **Dependencies**: Depends on check\_atlas\_status to ensure Atlas is accessible, as the ETL process sets up data assets to be tracked.
* **Role in Workflow**: Generates the data assets (database and table) that serve as the source for metadata ingestion, mimicking a real-world ETL pipeline.
* **Significance for Metadata Ingestion**: Creates the data entities (e.g., atlas\_test.employees) whose metadata (schema, lineage) will be ingested into Atlas, establishing the foundation for metadata tracking.

def simulate\_etl\_process():

    """Simulate an ETL process by creating and transforming Hive tables."""

    try:

        print("Starting ETL simulation...")

        # 1. source\_data table

        create\_hive\_table("""

            CREATE TABLE IF NOT EXISTS atlas\_test.source\_data (

                id INT,

                name STRING,

                age INT,

                department STRING,

                salary DOUBLE

            ) COMMENT 'Source employee data for ETL';

            INSERT INTO atlas\_test.source\_data

            SELECT \* FROM atlas\_test.employees;

        """)

        # 2. transformed\_data table with some transformations

        create\_hive\_table("""

            CREATE TABLE IF NOT EXISTS atlas\_test.transformed\_data (

                employee\_id INT,

                employee\_name STRING,

                department STRING,

                annual\_salary DOUBLE,

                is\_management BOOLEAN

            ) COMMENT 'Transformed employee data';

            INSERT INTO atlas\_test.transformed\_data

            SELECT

                id as employee\_id,

                name as employee\_name,

                department,

                salary \* 12 as annual\_salary,

                CASE WHEN salary > 80000 THEN true ELSE false END as is\_management

            FROM atlas\_test.source\_data;

        """)

        # 3. analytics\_data table (aggregated view)

        create\_hive\_table("""

            CREATE TABLE IF NOT EXISTS atlas\_test.analytics\_data (

                department STRING,

                avg\_salary DOUBLE,

                employee\_count INT,

                total\_annual\_cost DOUBLE

            ) COMMENT 'Analytics-ready department summary';

            INSERT INTO atlas\_test.analytics\_data

            SELECT

                department,

                AVG(salary) as avg\_salary,

                COUNT(\*) as employee\_count,

                SUM(salary \* 12) as total\_annual\_cost

            FROM atlas\_test.source\_data

            GROUP BY department;

        """)

        print("ETL simulation completed successfully")

        return True

    except Exception as e:

        print(f"Error in ETL simulation: {str(e)}")

        raise

def create\_hive\_table(hive\_commands):

    """Helper function to execute Hive commands."""

    try:

        with open('/tmp/hive\_etl\_commands.hql', 'w') as f:

            f.write(hive\_commands)

        beeline\_cmd = [

            "beeline",

            "-u", "jdbc:hive2://hive-server:10000",

            "-n", "hive",

            "-p", "hive",

            "-f", "/tmp/hive\_etl\_commands.hql"

        ]

        result = subprocess.run(beeline\_cmd, capture\_output=True, text=True)

        if result.returncode != 0:

            raise Exception(f"Hive command failed: {result.stderr}")

    except Exception as e:

        print(f"Error executing Hive commands: {str(e)}")

        raise

1. **Ingest Metadata to Atlas (**ingest\_metadata\_to\_atlas**)**

* **Type**: PythonOperator
* **Function**: ingest\_metadata\_to\_atlas()
* **Purpose**: Programmatically ingests metadata about the ETL process and data assets into Apache Atlas, capturing dataset properties, schema, and lineage.
* **Execution**: Initializes an Atlas client, generates a unique pipeline name with a timestamp, and calls create\_sample\_etl\_lineage to create metadata entities (e.g., database, table, and process entities) via the Atlas REST API. Handles success and failure cases with appropriate logging.
* **Dependencies**: Depends on simulate\_etl\_process to ensure the Hive data assets exist before ingesting their metadata.
* **Role in Workflow**: Transfers metadata about the atlas\_test.employees table and the simulated ETL process to Atlas, establishing entities and their relationships (e.g., lineage between source and transformed data).
* **Significance for Metadata Ingestion**: This task is the core of the metadata ingestion process, registering data assets and their lineage in Atlas for governance and discovery.

def ingest\_metadata\_to\_atlas():

    """Ingest metadata into Atlas using the direct API approach."""

    atlas\_url = "http://atlas:21000"

    pipeline\_name = f"etl\_pipeline\_{int(time.time())}"

    try:

        print(f"Ingesting metadata for ETL pipeline: {pipeline\_name}")

        # Create Atlas client

        atlas\_client = AtlasMetadataIngestion(atlas\_url)

        # Create the ETL lineage in Atlas

        success = create\_sample\_etl\_lineage(atlas\_client, pipeline\_name)

        if success:

            print("Successfully ingested metadata into Atlas")

            return True

        else:

            print("Failed to ingest metadata into Atlas")

            return False

    except Exception as e:

        print(f"Error ingesting metadata: {str(e)}")

        raise

1. **Verify Metadata in Atlas (**verify\_metadata\_in\_atlas**)**

* **Type**: PythonOperator
* **Function**: verify\_metadata\_in\_atlas()
* **Purpose**: Validates that the metadata ingested into Atlas is correct, complete, and includes expected entities and lineage relationships.
* **Execution**: Queries Atlas for specific entity types (e.g., hive\_database, hive\_table) and names (e.g., atlas\_test, employees), checks for lineage relationships, and validates metadata structure. Provides detailed feedback on any missing or incorrect elements.
* **Dependencies**: Depends on ingest\_metadata\_to\_atlas to ensure metadata has been ingested before verification.
* **Role in Workflow**: Ensures the integrity of the ingested metadata, confirming that Atlas accurately reflects the ETL process and data assets.
* **Significance for Metadata Ingestion**: Validates the success of the ingestion process, ensuring that metadata is reliable and usable for governance purposes.

def verify\_metadata\_in\_atlas():

    """Verify that metadata has been properly ingested into Atlas."""

    atlas\_url = "http://atlas:21000"

    auth = ("admin", "admin")

    headers = {"Content-Type": "application/json", "Accept": "application/json"}

    # Atlas client

    atlas\_client = AtlasMetadataIngestion(atlas\_url)

    # entities we expect to find

    expected\_entities = [

        {"type": "hive\_db", "name": "atlas\_test"},

        {"type": "hive\_table", "name": "source\_data"},

        {"type": "hive\_table", "name": "transformed\_data"},

        {"type": "hive\_table", "name": "analytics\_data"}

    ]

    # Check each expected entity

    missing\_entities = []

    for entity in expected\_entities:

        found\_entity = atlas\_client.get\_entity\_by\_attribute(entity["type"], "name", entity["name"])

        if not found\_entity:

            missing\_entities.append(f"{entity['type']}:{entity['name']}")

    if missing\_entities:

        print(f"The following entities are missing in Atlas: {', '.join(missing\_entities)}")

        return False

    print("All expected entities are present in Atlas")

    # Check for lineage - verify the transformed\_data table has lineage

    transformed\_entity = atlas\_client.get\_entity\_by\_attribute("hive\_table", "name", "transformed\_data")

    if transformed\_entity:

        table\_guid = transformed\_entity['guid']

        lineage\_url = f"{atlas\_url}/api/atlas/v2/lineage/{table\_guid}"

        lineage\_response = requests.get(lineage\_url, auth=auth)

        if lineage\_response.status\_code == 200:

            lineage\_data = lineage\_response.json()

            relations = lineage\_data.get('relations', {})

            if relations and len(relations) > 0:

                print("Lineage information is available!")

                return True

            else:

                print("Lineage information is missing or incomplete")

    return False

#### **Integrations for metadata ingestion in Apache Atlas**

1. **Hive Metastore**: The DAG interacts with Hive via Beeline to create and manage data assets, which are then tracked by Atlas through its integration with the Hive metastore. This ensures that metadata (e.g., table schemas) is automatically captured.The integration is configured through hive-site.xml and environment variables.

# hive-site.xml

-----------------

<property>

    <name>hive.exec.post.hooks</name>

    <value>org.apache.atlas.hive.hook.HiveHook</value>

</property>

<property>

    <name>atlas.hook.hive.synchronous</name>

    <value>true</value>

</property>

<property>

    <name>atlas.rest.address</name>

    <value>http://atlas:21000</value>

</property>

The Hive hook automatically captures:

* Table schemas
* Column definitions and data types
* Table statistics
* Data lineage

def create\_hive\_table():

    """For creating a sample Hive table for testing Atlas integration using HiveServer2."""

    try:

        # Hive commands to create table and load data

        hive\_cmd = """

        CREATE DATABASE IF NOT EXISTS atlas\_test;

        DROP TABLE IF EXISTS atlas\_test.employees;

        CREATE TABLE atlas\_test.employees (

            id INT,

            name STRING,

            age INT,

            department STRING,

            salary DOUBLE

        ) COMMENT 'Employee data for Atlas integration test';

        INSERT INTO atlas\_test.employees VALUES

            (1, 'John Doe', 45, 'Engineering', 85000),

            (2, 'Jane Smith', 22, 'Marketing', 62000),

            (3, 'Robert Johnson', 17, 'Intern', 25000),

            (4, 'Lisa Williams', 29, 'Sales', 71000),

            (5, 'Michael Brown', 19, 'Intern', 27000),

            (6, 'Sarah Miller', 38, 'Finance', 92000),

            (7, 'David Wilson', 16, 'Intern', 24000),

            (8, 'Jennifer Taylor', 31, 'Human Resources', 68000),

            (9, 'William Davis', 55, 'Executive', 120000),

            (10, 'Emily Anderson', 27, 'Engineering', 75000);

        """

        # Writing Hive commands to a file

        with open('/tmp/hive\_commands.hql', 'w') as f:

            f.write(hive\_cmd)

        # Hive commands execution using beeline

        beeline\_cmd = [

            "beeline",

            "-u", "jdbc:hive2://hive-server:10000",

            "-n", "hive",

            "-p", "hive",

            "-f", "/tmp/hive\_commands.hql"

        ]

1. **Atlas REST API**: The ingest\_metadata\_to\_atlas and verify\_metadata\_in\_atlas tasks use the Atlas REST API (<http://atlas:21000>) to create and query metadata entities, enabling programmatic metadata management.

Def def ingest\_metadata\_to\_atlas():

    """Ingest metadata into Atlas using the direct API approach."""

    atlas\_url = "http://atlas:21000"

    pipeline\_name = f"etl\_pipeline\_{int(time.time())}"

    try:

        print(f"Ingesting metadata for ETL pipeline: {pipeline\_name}")

        # Create Atlas client

        atlas\_client = AtlasMetadataIngestion(atlas\_url)

        # Create the ETL lineage in Atlas

        success = create\_sample\_etl\_lineage(atlas\_client, pipeline\_name)

        if success:

            print("Successfully ingested metadata into Atlas")

            return True

        else:

            print("Failed to ingest metadata into Atlas")

            return False

    except Exception as e:

        print(f"Error ingesting metadata: {str(e)}")

        raise

def verify\_metadata\_in\_atlas():

    """Verify that metadata has been properly ingested into Atlas."""

    atlas\_url = "http://atlas:21000"

    auth = ("admin", "admin")

    headers = {"Content-Type": "application/json", "Accept": "application/json"}

    # Atlas client

    atlas\_client = AtlasMetadataIngestion(atlas\_url)

1. **Lineage Tracking**: The workflow captures ETL lineage (e.g., from source data to the employees table), which is critical for data governance and understanding data provenance in Atlas.

def create\_sample\_etl\_lineage(atlas\_client, pipeline\_name="sample\_etl"):

    """Creating a sample ETL pipeline with lineage in Atlas."""

    # Step 1: database creation

    db\_guid = atlas\_client.create\_hive\_db("atlas\_test")

    if not db\_guid:

        logger.error("Failed to create database. Exiting.")

        return False

    # Step 2: source table creation with columns

    source\_columns = [

        {"name": "id", "type": "int", "comment": "Primary identifier"},

        {"name": "name", "type": "string", "comment": "Person name"},

        {"name": "department", "type": "string", "comment": "Department name"},

        {"name": "salary", "type": "double", "comment": "Raw salary amount"}

    ]

    source\_table\_guid = atlas\_client.create\_hive\_table(

        "source\_data",

        "atlas\_test",

        db\_guid,

        description="Raw source data for ETL processing",

        columns=source\_columns

    )

    # ETL process cration for source to transformed (Extract + Transform)

    extract\_transform\_guid = atlas\_client.create\_etl\_process(

        f"{pipeline\_name}\_extract\_transform",

        [source\_table\_guid],

        [transformed\_table\_guid],

        description="Extract and transform process"

    )

1. **Error Handling**: Comprehensive error handling ensures that failures (Atlas connectivity issues, Hive SQL errors) are caught early, with clear error messages for debugging.

def check\_connection(self):

    """Check if Atlas is running and return version info."""

    try:

        response = requests.get(f"{self.atlas\_url}/api/atlas/admin/version", auth=self.auth)

        if response.status\_code == 200:

            version\_info = response.json()

            logger.info(f"Successfully connected to Atlas {version\_info.get('Version', 'Unknown')}")

            return True

        else:

            logger.error(f"Atlas returned status code {response.status\_code}")

            return False

    except Exception as e:

        logger.error(f"Error connecting to Atlas: {e}")

        return False

def create\_hive\_table(self, table\_name, db\_name, db\_guid=None, cluster\_name="primary",

                    description=None, owner="admin", columns=None):

    """Create a Hive table entity in Atlas."""

    try:

        response = requests.post(

            f"{self.atlas\_url}/api/atlas/v2/entity",

            data=json.dumps(entity),

            headers=self.headers,

            auth=self.auth

        )

        if response.status\_code == 200:

            result = response.json()

            guid = result.get('guidAssignments', {}).values()

            guid = list(guid)[0] if guid else None

            logger.info(f"Created Hive table '{qualified\_name}' with GUID: {guid}")

            return guid

        else:

            logger.error(f"Failed to create Hive table: {response.status\_code} - {response.text}")

            return None

    except Exception as e:

        logger.error(f"Error creating Hive table: {e}")

        return None

#### **Challenges for metadata ingestion in Apache Atlas**

1. **Atlas Connectivity**: As seen in prior errors (Connection refused to datahub-gms:8080), similar issues can occur with Atlas if the server (http://atlas:21000) is not running or accessible. The check\_atlas\_status task mitigates this by validating connectivity upfront.
2. **Hive Integration**: Misconfigurations in the Hive metastore or Beeline connection (e.g., incorrect JDBC URL, credentials) could prevent the simulate\_etl\_process task from succeeding.
3. **Resource Constraints**: Insufficient resources for Airflow, Hive, or Atlas containers could cause failures, especially for metadata ingestion tasks that require API calls and database interactions.
4. **Metadata Completeness**: The ingest\_metadata\_to\_atlas task must include all relevant metadata (e.g., entity types, lineage relationships) to ensure Atlas provides a comprehensive view of the data assets.

# Chapter 7 Findings and Analysis

## 7.1 Summary of Findings

The Proof of Concept (POC) evaluations of Apache Atlas, DataHub, and Amundsen revealed distinct strengths and limitations in their metadata management capabilities. Apache Atlas demonstrated robust governance and lineage tracking, particularly for Hadoop ecosystems, with automatic metadata capture via Hive hooks for employee data processing, achieving ingestion times of ~2.5 seconds per entity. Its comprehensive lineage visualization and PII tagging support compliance needs, DataHub excelled in user-friendly metadata exploration and modern data stack integration, with a scalable architecture using PostgreSQL, MySQL, and Elasticsearch, and efficient ingestion via Airflow DAGs. However, its lineage capabilities are less advanced, lacking deep column-level support, and dependency management issues like startup order errors posed challenges.

Amundsen’s POC was hampered by persistent metadata ingestion failures and partial UI loading, attributed to Elasticsearch indexing errors, Neo4j connection issues. Despite its potential for intuitive data discovery via a Flask/React frontend, Amundsen’s modular architecture, its complex setup and limited governance features limit its competitiveness, with slowed community development prompting exploration of alternatives like OpenMetadata. The workflows for Atlas and DataHub showcased robust ETL and metadata ingestion processes, with Atlas leveraging Hive integration and DataHub focusing on PostgreSQL and GMS API connectivity, though both faced connectivity issues (e.g., “Connection refused” errors) that required health checks and network tuning.

Capability comparisons underscored Atlas’s governance and lineage in Hadoop environments, while DataHub prioritized usability. Amundsen’s discovery-focused design was unfulfilled due to technical failures, highlighting its dependency on stable indexing and community support. These findings indicate that Atlas suits governance-heavy, Hadoop-centric use cases, DataHub is ideal for modern, user-friendly deployments, and Amundsen requires significant troubleshooting to realize its discovery potential.

## 7.2 Challenges in Metadata Management

Data silos pose a significant challenge in metadata management, as demonstrated in the POCs where Atlas’s Hadoop-centric integration limited its effectiveness for non-Hadoop systems like PostgreSQL, and DataHub’s focus on modern stacks created gaps with legacy systems. These silos hinder comprehensive metadata catalogs, reducing discoverability and lineage accuracy, as seen in Amundsen’s unindexed metadata due to ingestion failures. Mitigation strategies include adopting interoperable frameworks like the Open Metadata Standard and leveraging extensible APIs to bridge systems, alongside data mesh principles to promote domain-driven metadata ownership.

Inconsistent metadata standards further complicate integration and governance. DataHub’s POC revealed issues with incorrect URNs causing metadata visibility problems, while Atlas’s flexible entity types risked inconsistency without enforced policies. These inconsistencies reduce trust and usability, exacerbated by Amundsen’s misaligned configurations disrupting metadata indexing. Implementing organization-wide standards, such as consistent URN formats and classification schemas, and automating validation processes can enhance consistency across tools like Atlas and DataHub, with complementary tools like Collibra providing standardized governance frameworks.

# Chapter 8 Conclusion

Metadata management is pivotal for enabling data governance, discoverability, and compliance in modern data architectures, acting as the backbone for organizations navigating complex, distributed data ecosystems. The POC evaluations of Apache Atlas, DataHub, and Amundsen revealed distinct strengths and challenges, providing insights into their suitability for various use cases. Apache Atlas excels in governance-heavy environments, particularly within Hadoop ecosystems, with robust lineage tracking and PII tagging demonstrated in the **atlas\_integration\_final.py** DAG, though its scalability issues and complex UI hinder broader adoption. DataHub offers user-friendly metadata exploration and seamless integration with modern stacks like PostgreSQL, as shown in the **csv\_etl\_pipeline** DAG, but its limited column-level lineage and dependency management challenges require careful configuration. These findings emphasize that effective metadata management requires aligning tool capabilities with organizational needs, ensuring scalability, and fostering user adoption to maximize data-driven decision-making.

Amundsen, intended for intuitive data discovery, faced significant setbacks in the POC due to metadata ingestion failures and UI issues, attributed to Elasticsearch indexing errors, Neo4j connection refusals, and insufficient Docker memory. Recent web sources indicate Amundsen has matured as a system, with a stable architecture hosted by the LF AI & Data Foundation, but its community has become less active, with slowed development and limited updates since 2021, as evidenced by stale GitHub issues and reduced community engagement on Slack. Version incompatibility issues, such as outdated Jira API integrations and SQLAlchemy dependencies, further complicate deployments, requiring manual fixes like pinning SQLAlchemy to 1.3.23. These challenges highlight the risks of relying on tools with minimal community support, underscoring the need for robust, actively maintained solutions in metadata management.

The report’s insights reveal that metadata management is not merely a technical exercise but a strategic imperative for organizations seeking to enhance data visibility, compliance, and self-service analytics. Atlas and DataHub demonstrate viable paths forward, with Atlas suiting regulated, Hadoop-centric environments and DataHub catering to modern, user-focused deployments. Amundsen’s potential remains unrealized due to technical and community-related barriers, suggesting a shift toward alternatives like DataHub or OpenMetadata. By addressing data silos, standardizing metadata, and ensuring scalability, organizations can leverage metadata management to transform raw data into actionable insights, driving operational efficiency and competitive advantage.

# References

Apache Software Foundation. (n.d.). *Apache Atlas: Data governance and metadata framework for Hadoop*. <https://atlas.apache.org>

Atlan. (2022, September 14). *Amundsen alternatives: DataHub, Metacat, and Apache Atlas*. <https://atlan.com/atlassian/amundsen-alternatives/>

DataHub. (2021). *DataHub release 0.8.20 notes*. <https://github.com/datahub-project/datahub/releases/tag/v0.8.20>

www.amundsen.io. (n.d.). *Amundsen: Overview*. <https://www.amundsen.io>

GitHub. (n.d.). apache/atlas: Apache Atlas - Open metadata management and governance. <https://github.com/apache/atlas>

DataHub. (n.d.). *DataHub: Quickstart guide*. <https://datahubproject.io/docs/quickstart/>

Atlan. (2023, December 4). *Apache Atlas: Origin, architecture & features guide (2025)*. <https://atlan.com/atlas/atlas-data-guide/>

Apache Atlas. (n.d.). *Apache Atlas: Hook & bridge for Apache Hive*. <https://atlas.apache.org/#/Hook/Bridge-Hive>

DataHub. (n.d.). *DataHub: Metadata ingestion source for PostgreSQL*. <https://datahubproject.io/docs/generated/ingestion/sources/postgres/>

# Appendices

1. **Apache Atlas metadata management snapshots**

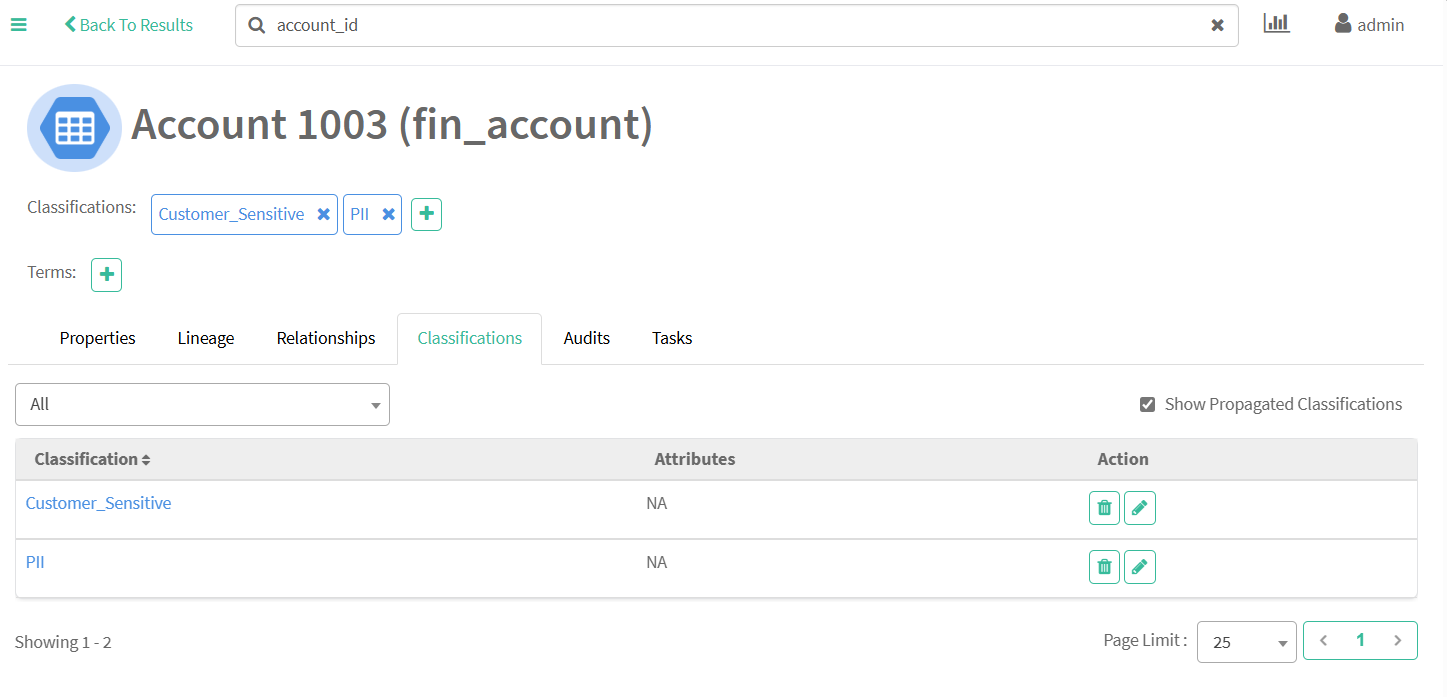
****

Figure I. 1 Entity Detail for Account with its classification

The image highlights classifications such as "Customer\_Sensitive" and "PII" with their respective attributes, along with navigation tabs for Properties, Lineage, Relationships, Classifications, Audits, and Tasks. The interface includes options to show propagated classifications and manage page limits for displayed entries.

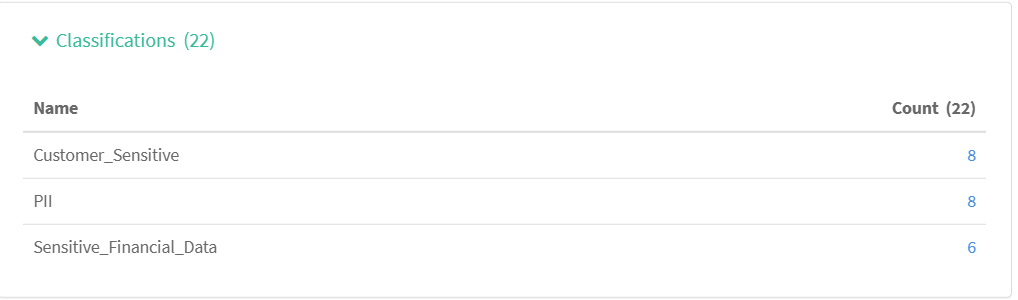


Figure I. 2 Summary of Data Classifications and their counts

The image displays an overview of classification categories such as Customer\_Sensitive, PII, and Sensitive\_Financial\_Data, along with the count of associated entities.

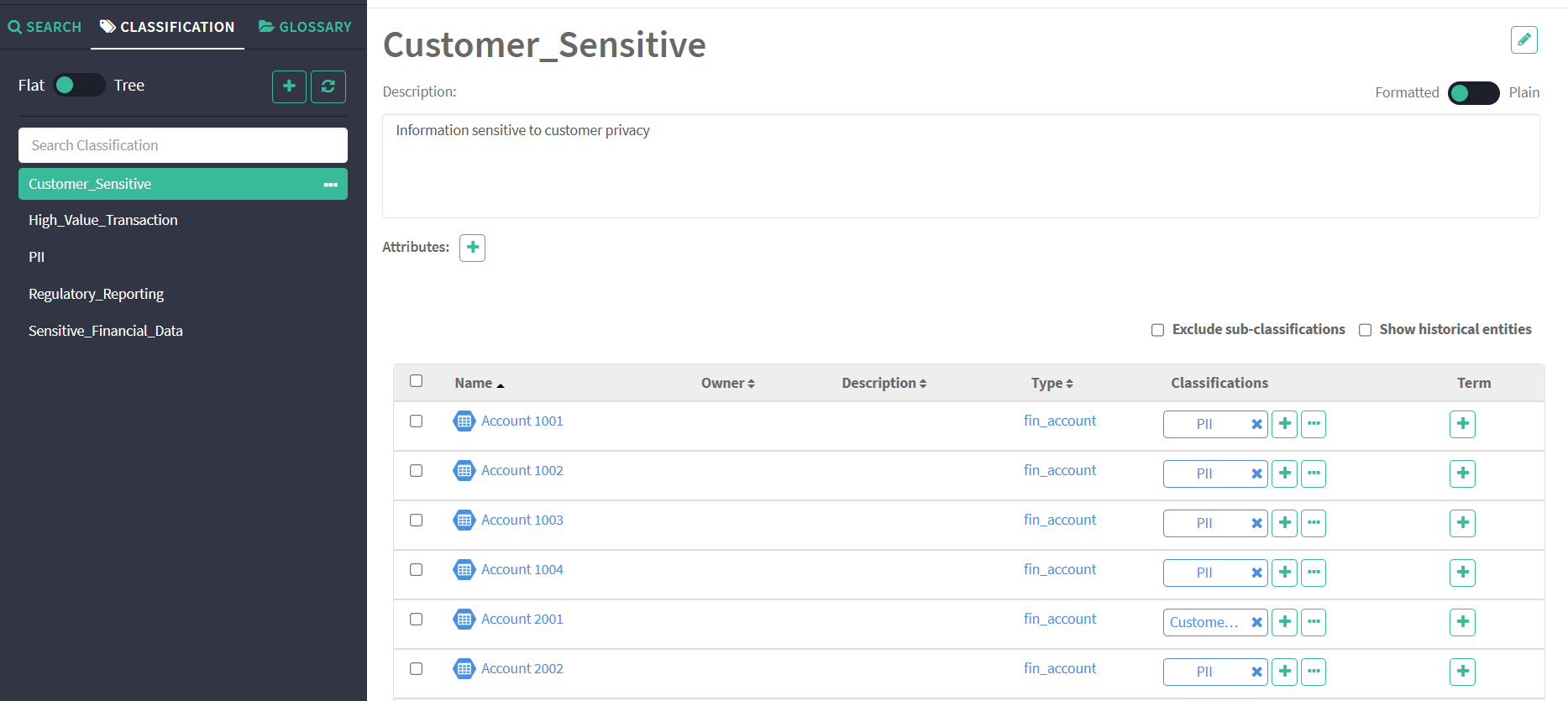


Figure I. 3 Customer Sensitive Data Classification Dashboard

The image shows the Apache Atlas classification management interface, specifically highlighting the "Customer\_Sensitive" classification. The left sidebar lists data classification types: Customer\_Sensitive (selected), High\_Value\_Transaction, PII, Regulatory\_Reporting, and Sensitive\_Financial\_Data. The main panel details the Customer\_Sensitive classification, defined as "Information sensitive to customer privacy." It includes a table of financial accounts (Account 1001-1004, 2001-2002), all categorized as "fin\_account" type, with classification tags "PII" and "Customer..." indicating customer-sensitive financial records. The interface provides options for bulk operations, filtering, managing sub-classifications, and viewing historical entities within the data governance framework.

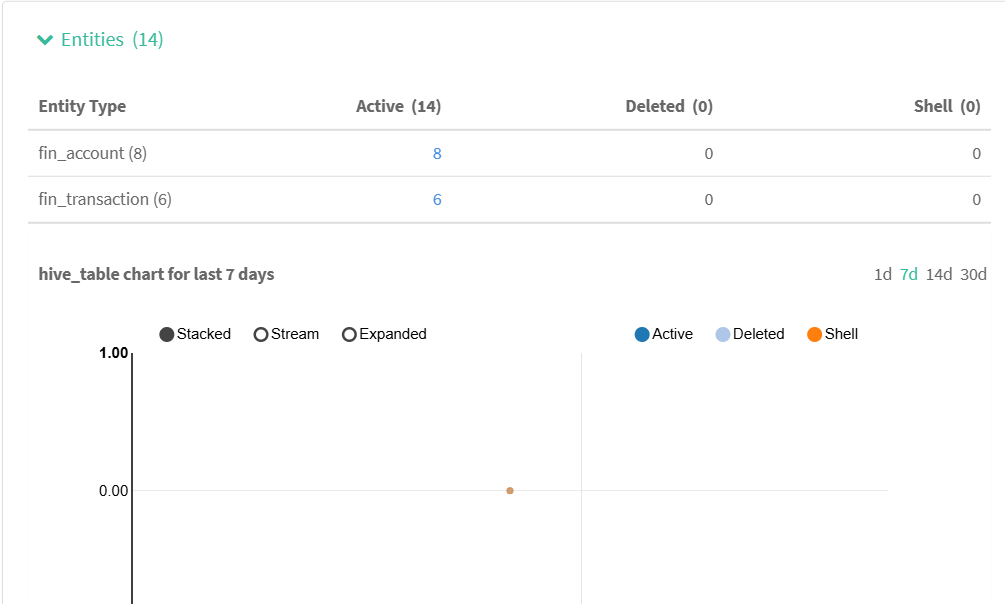


Figure I. 4 Entity Type Distribution and Activity Status

The image illustrates the number of active entities categorized under fin\_account and fin\_transaction types, along with a historical activity trend.

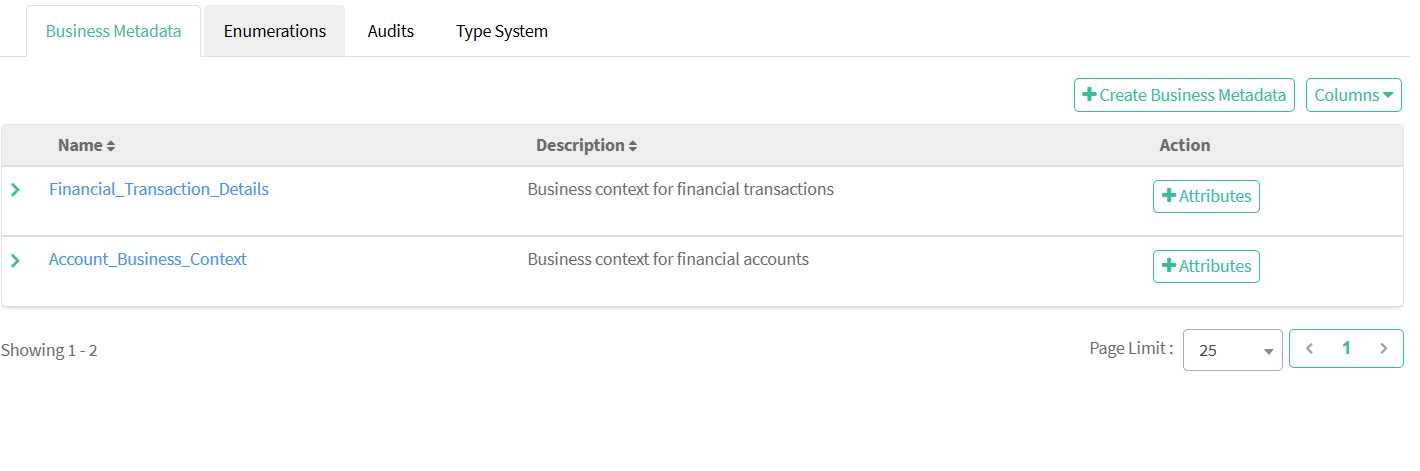


Figure I. 5 Business Metadata Management Interface

The image shows the Business Metadata tab in Apache Atlas, focused on configuring and managing business context metadata definitions. It displays two business metadata types: "Financial\_Transaction\_Details" (defined as "Business context for financial transactions") and "Account\_Business\_Context" (defined as "Business context for financial accounts"). Each type has an expandable row with an "Attributes" button for detailed configuration. The top navigation includes tabs for Business Metadata (selected), Enumerations, Audits, and Type System. A "Create Business Metadata" button allows adding new metadata definitions, and column management options are available. This interface supports defining business-specific metadata schemas to enhance data governance and provide meaningful context to technical data assets.

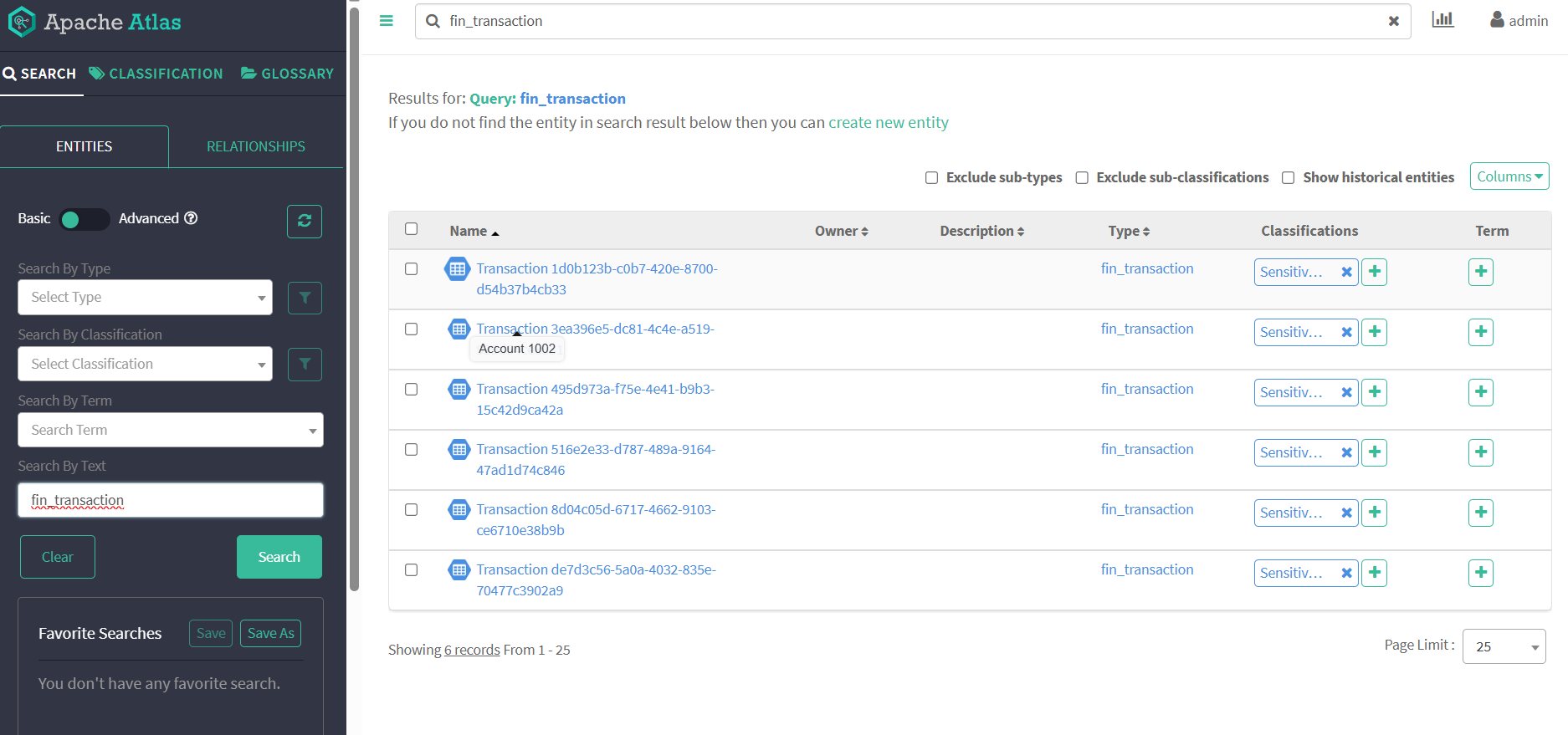


Figure I. 6 Financial Transaction Search Results Interface

The image displays the Apache Atlas search results page for the query "fin\_transaction." It shows six financial transaction records, each with unique transaction IDs, classified as "fin\_transaction" type and tagged with "Sensitive..." classification labels. The left sidebar includes advanced search filters for Type, Classification, Term, and Text, with "fin\_transaction" entered in the text search field. The results table lists transaction names, unique identifiers (e.g., **Transaction 8d04c05d-6717-4662-9103-ce6710e38b9b**), and columns for Owner, Description, Type, Classifications, and Terms. Options are available for excluding sub-types and sub-classifications, displaying historical entities, and managing column visibility, highlighting Apache Atlas's metadata search and discovery features within its data governance platform.

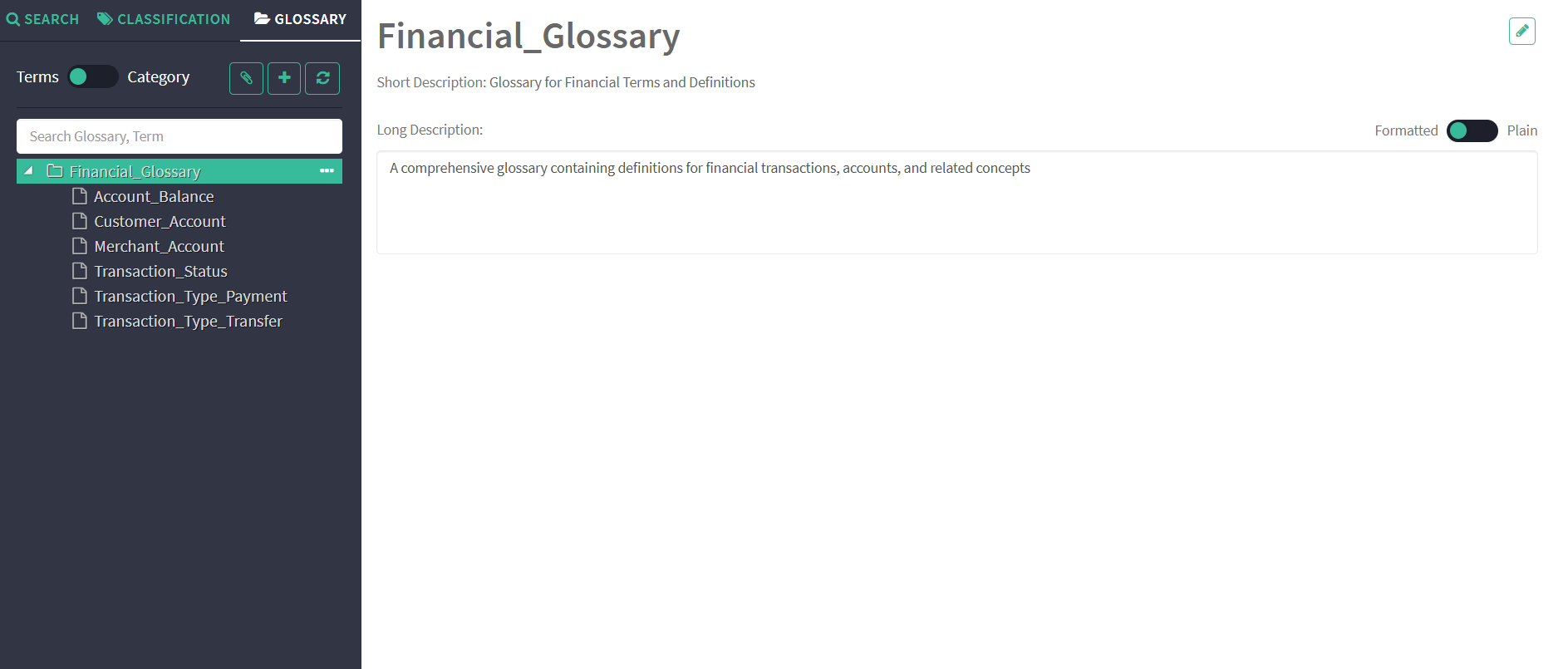


Figure I. 7 Financial Glossary Management Interface

The image displays the Apache Atlas Glossary section, highlighting the "Financial\_Glossary" as a repository for financial terminology and definitions.

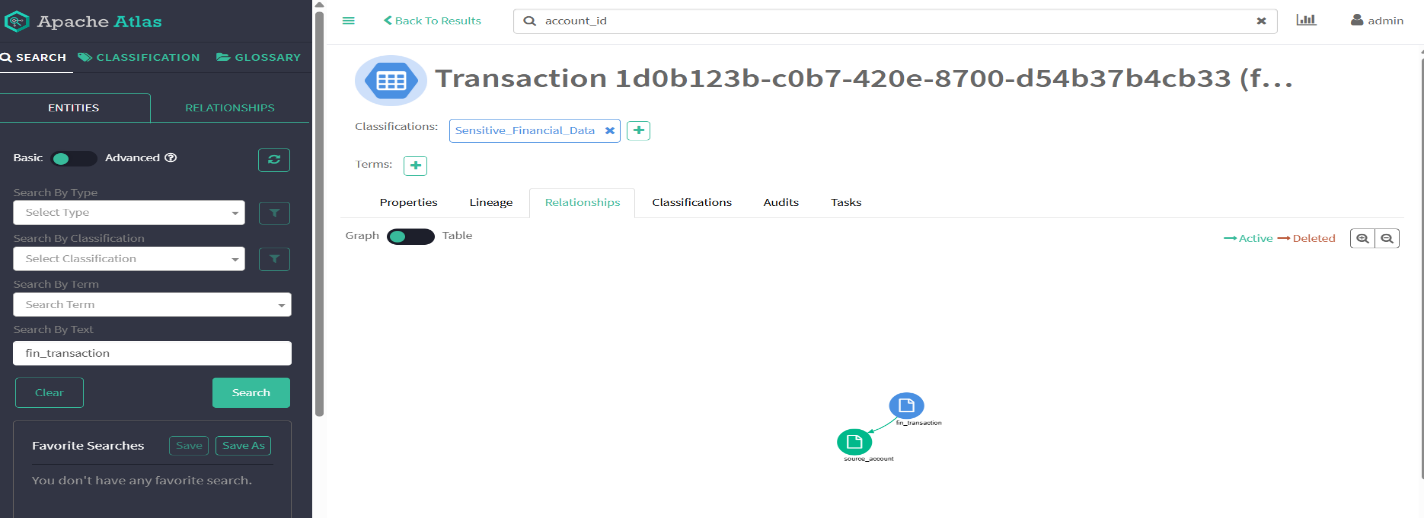


Figure I. 8 Transaction Relationships Data Lineage View

The image displays the Relationships tab for a financial transaction (Transaction 1d0b123b-c0b7-420e-8700-d54b37b4cb33) in Apache Atlas, showcasing its data lineage and entity relationships. The transaction is classified as "Sensitive\_Financial\_Data." The main content area presents a lineage diagram with two nodes: a blue "fin\_transaction" node connected to a green "source\_account" node, illustrating the relationship between the transaction and its source account.

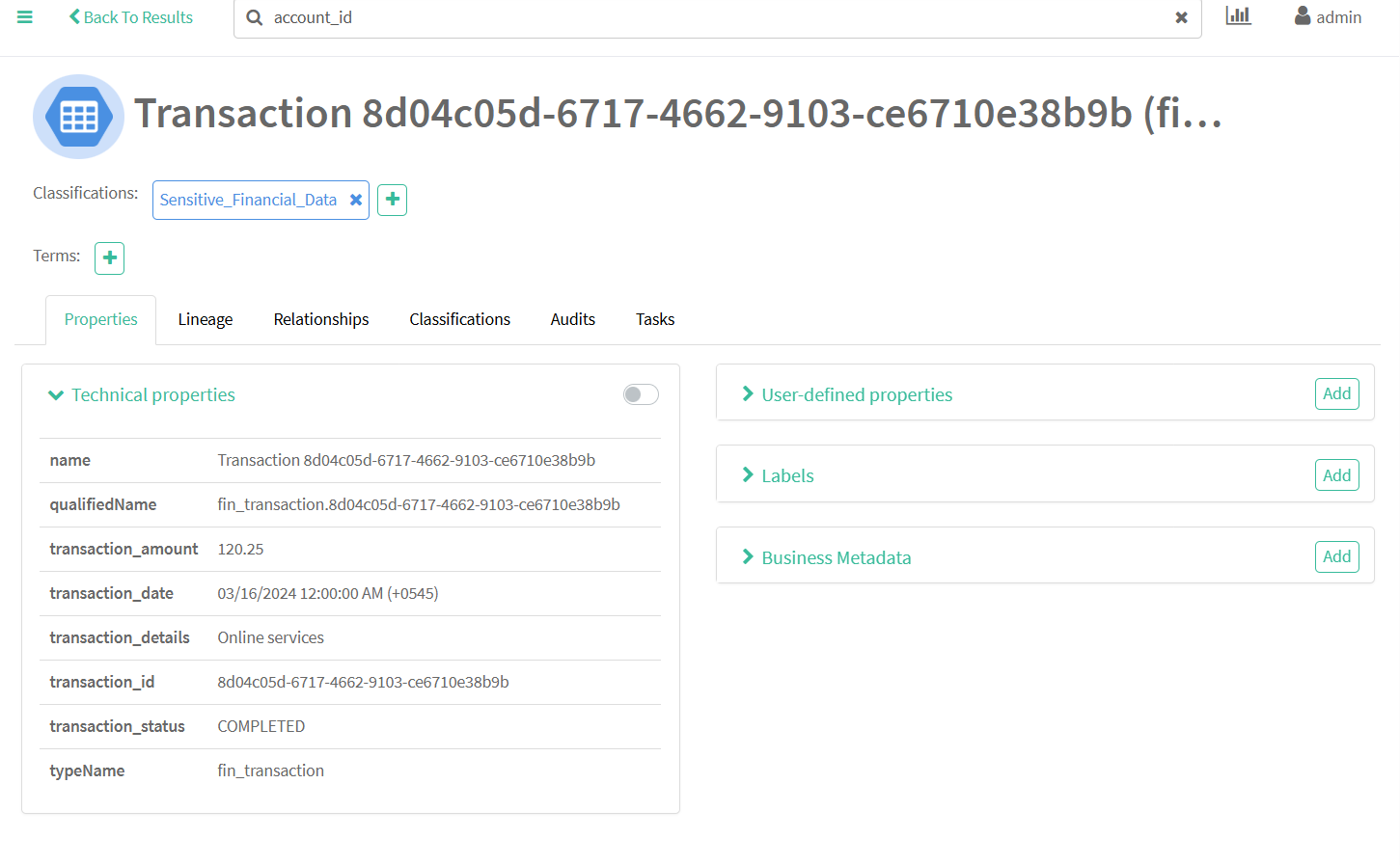


Figure I. 9 Financial Transaction Metadata View

The image displays a detailed metadata view of a financial transaction record in Apache Atlas, showing technical properties: transaction ID (8d04c05d-6717-4662-9103-ce6710e38b9b), amount ($120.25), date (03/16/2024), and status (COMPLETED). The interface includes metadata management tabs: Properties (selected), Lineage, Relationships, Classifications, Audits, and Tasks. The transaction is classified as "Sensitive\_Financial\_Data" and typed as "fin\_transaction" for online services. Expandable sections for user-defined properties, labels, and business metadata are present, demonstrating Apache Atlas's role in tracking and managing financial transaction metadata within a data governance system.

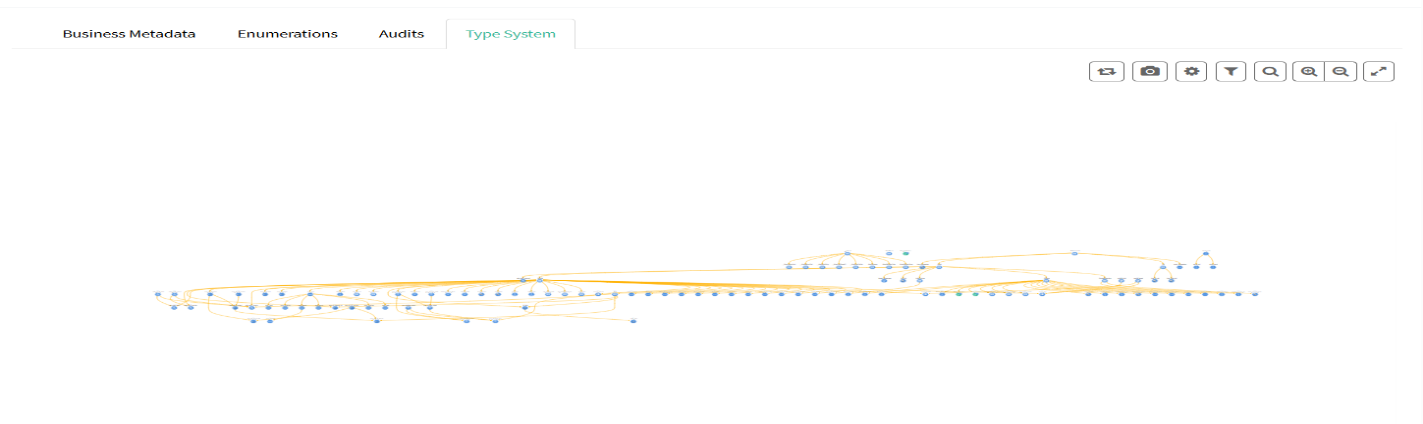


Figure I. 10 Type System Network Visualization

The image displays the Type System tab in Apache Atlas, showcasing a network visualization of metadata type relationships and dependencies. The graph features multiple blue nodes connected by yellow/orange edges, representing relationships between entity types, attribute types, and classification types. The visualization shows a hierarchical or clustered structure with denser connections in specific areas, indicating closely related type definitions. The top navigation includes tabs for Business Metadata, Enumerations, Audits, and Type System (selected), with visualization control icons (zoom, filter, search, and layout options) in the top-right toolbar. This interface enables administrators and data architects to understand the structure and interdependencies of the metadata type system, central to Apache Atlas's organization and categorization of metadata entities within its data governance framework.

1. **DataHub metadata management snapshots**

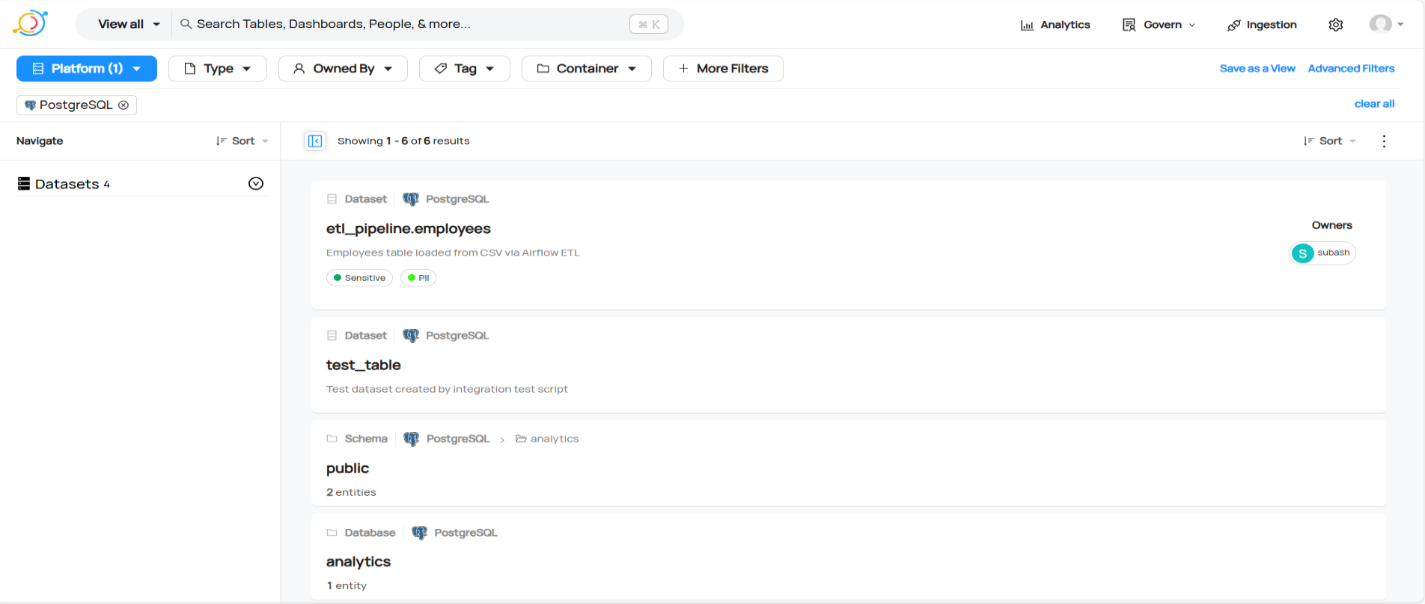
****

Figure II. 1 DataHub - PostgreSQL Platform Data Catalog Overview

The image displays DataHub's main catalog interface, showcasing ingested PostgreSQL metadata. The platform filter is set to PostgreSQL, displaying six results: datasets like "etl\_pipeline.employees" (an employee table loaded from CSV via Airflow ETL, tagged as Sensitive and PII), "test\_table" (a test dataset), the "public" schema with two entities, and the "analytics" database with one entity. The interface includes filtering options for Platform, Type, Owned By, Tag, and Container, along with search and navigation tools. This demonstrates DataHub's ability to organize and present database metadata in a user-friendly catalog format with clear tagging and ownership details.

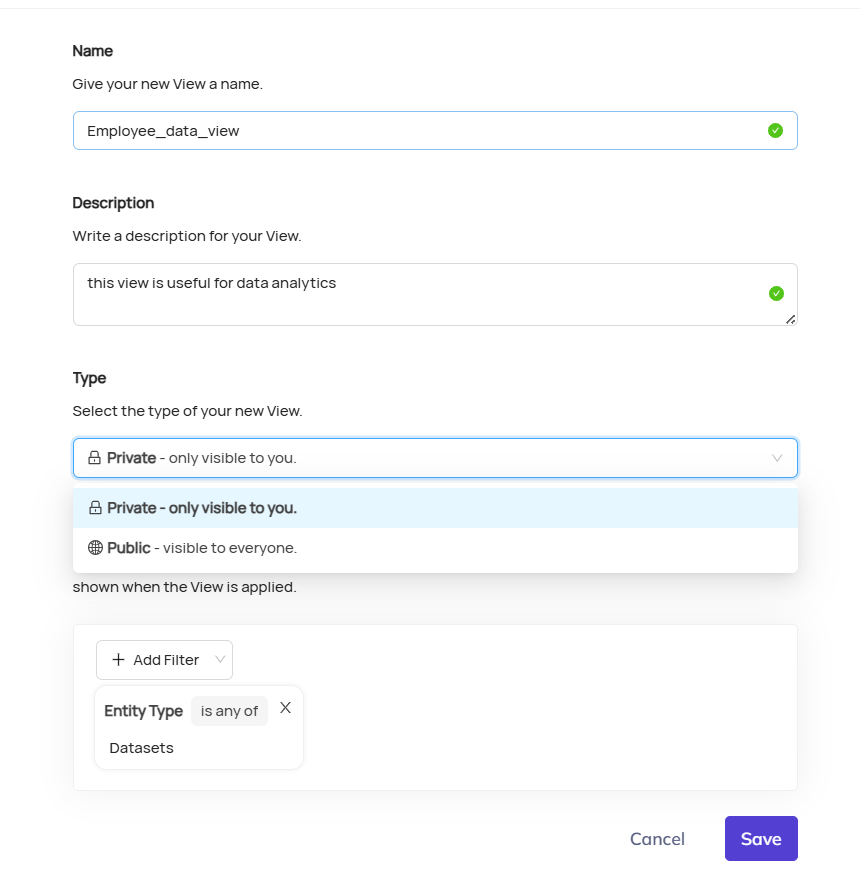


Figure II. 2 Custom View Creation Dialog

The image displays DataHub's view creation interface for generating custom filtered views of the data catalog. The form includes fields for a view name ("Employee\_data\_view"), a description ("this view is useful for data analytics"), and visibility type (Private or Public). Filtering options are set to show "Datasets" as the entity type. This feature allows users to create personalized or team-specific views of the data catalog, enabling focused access to relevant datasets and metadata based on specific use cases or organizational needs.

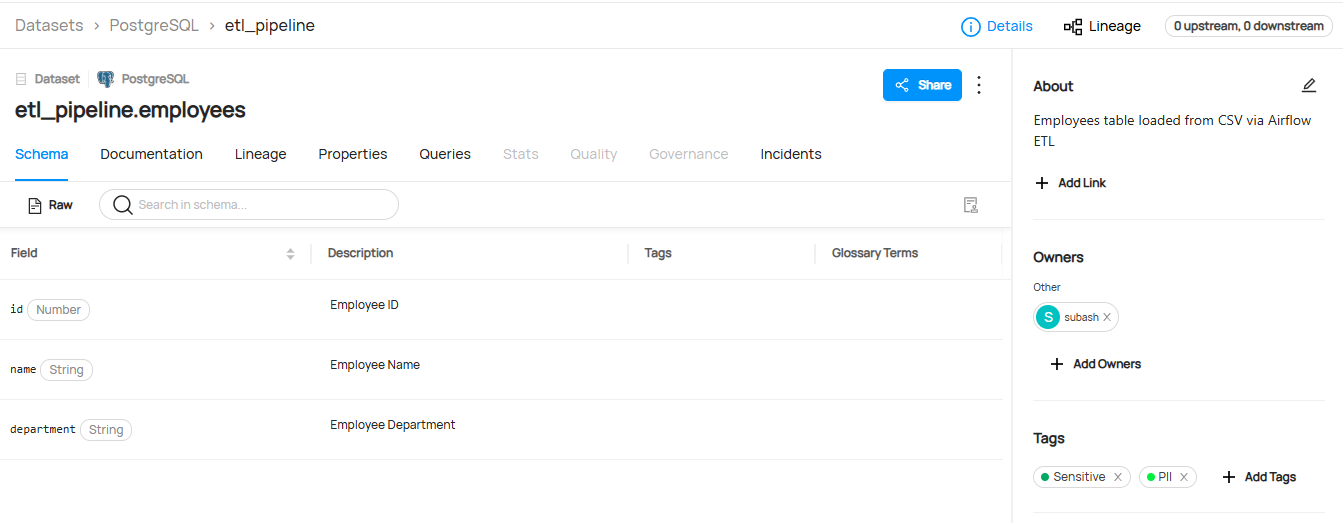


Figure II. 3 Dataset Schema View with Field-Level Metadata

This image displays DataHub's detailed schema view for the "etl\_pipeline.employees" PostgreSQL dataset. The interface shows the table structure with three fields: "id" (Number/Employee ID), "name" (String/Employee Name), and "department" (String/Employee Department). Each field includes data type information and descriptive labels. The right panel shows comprehensive metadata including ownership (assigned to user "subash"), data sensitivity tags ("Sensitive" and "PII"), and contextual information noting this is an "Employees table loaded from CSV via Airflow ETL". The interface provides multiple tabs for exploring different aspects of the dataset including Schema, Documentation, Lineage, Properties, Queries, Stats, Quality, Governance, and Incidents, demonstrating DataHub's comprehensive data governance capabilities.

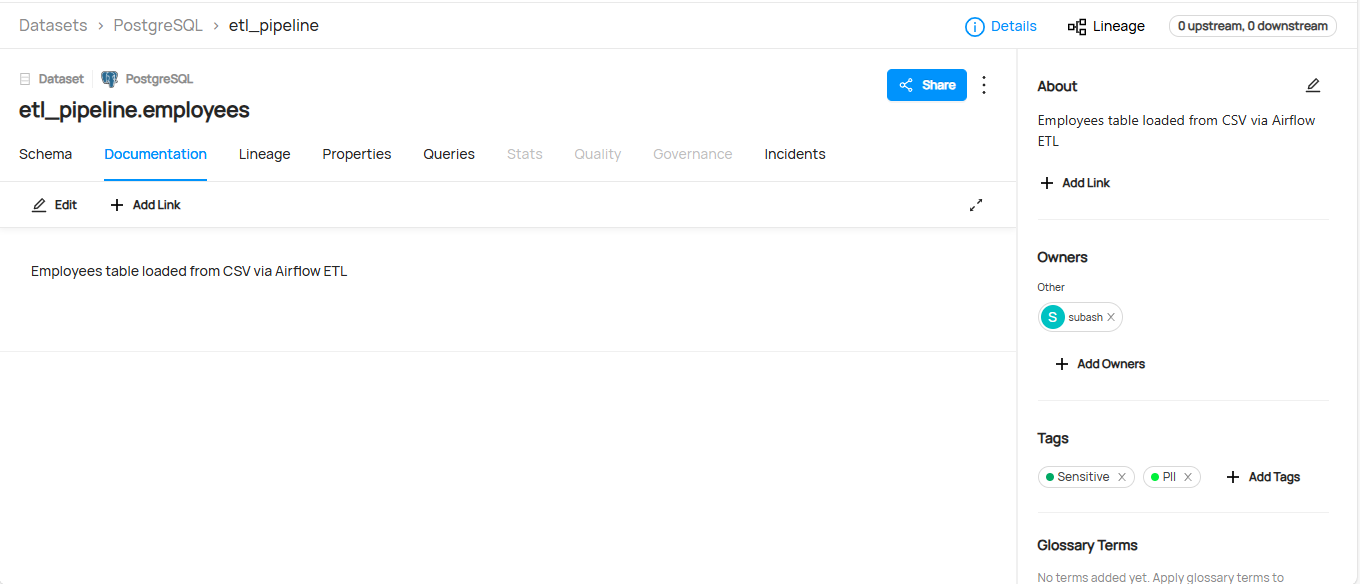


Figure II. 4 Dataset Documentation Tab View

The image displays DataHub's Documentation tab for the "etl\_pipeline.employees" dataset. The main content area shows editable documentation text stating "Employees table loaded from CSV via Airflow ETL," with options to Edit and Add Link. The Documentation tab is selected among available tabs (Schema, Documentation, Lineage, Properties, etc.). The right sidebar lists metadata, including ownership (subash), data classification tags (Sensitive, PII44), and an empty glossary terms section. This interface highlights DataHub's ability to maintain detailed documentation alongside technical metadata, facilitating the capture and sharing of business context, data lineage, and usage guidelines for datasets.

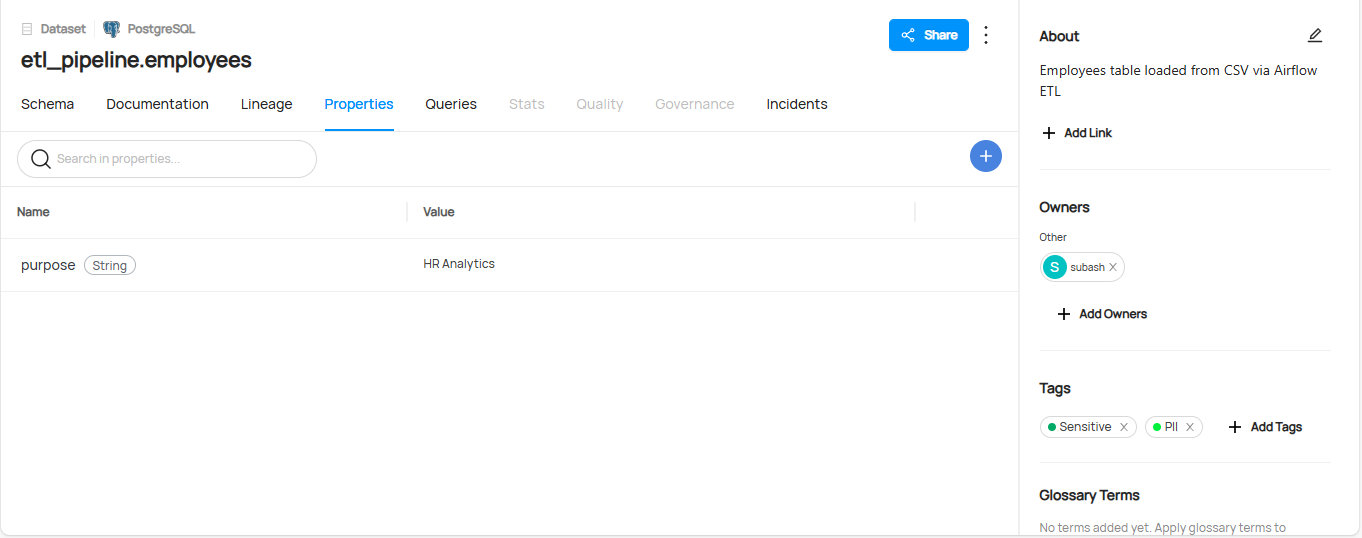


Figure II. 5 Dataset Properties Configuration View

The image displays DataHub's Properties tab for the "etl\_pipeline.employees" dataset. The interface presents a searchable table with Name and Value columns, showing one custom property: "purpose" (String type) with the value "HR Analytics." A plus button allows adding new properties, and a search function enables filtering.

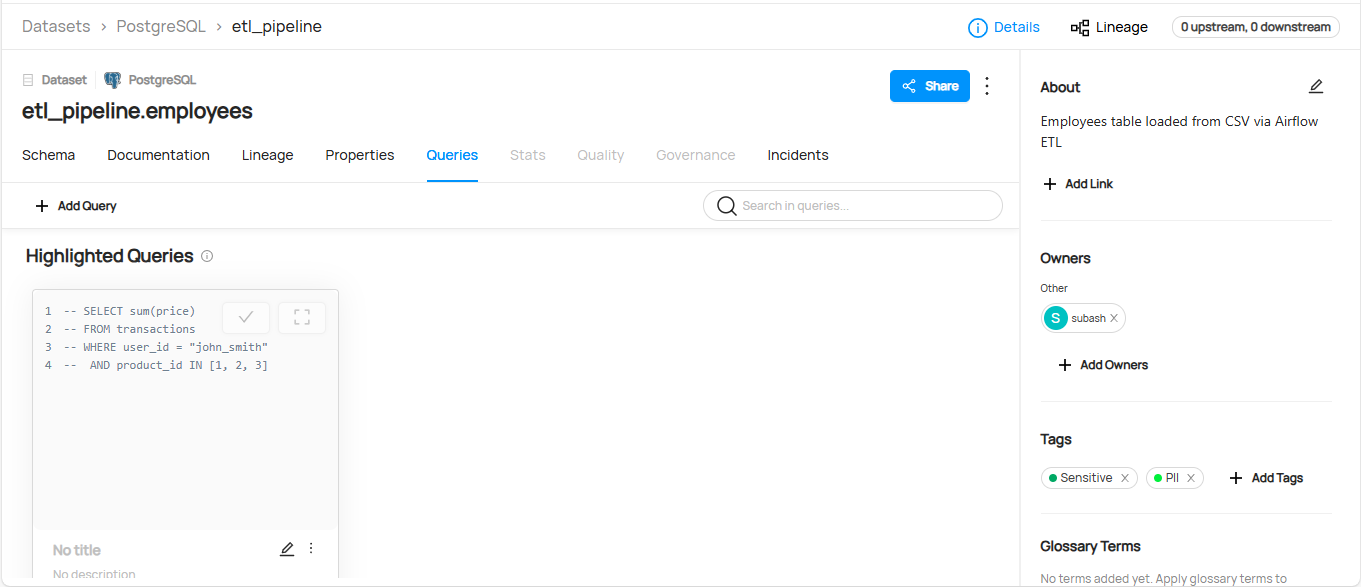


Figure II. 6 SQL Queries Tab with Highlighted Query Examples

The image displays DataHub's Queries tab for the "etl\_pipeline.employees" dataset, showcasing highlighted SQL queries executed against the table. A sample SQL query with syntax highlighting includes SELECT, FROM, WHERE, and AND clauses, filtering transactions by user\_id "john\_smith" and product\_id values, indicating it may relate to a transactions table rather than the employees table. The interface provides options to "Add Query," search queries, and edit query metadata (currently listed as "No title" and "No description"). This feature enables users to discover common query patterns and understand typical dataset usage.

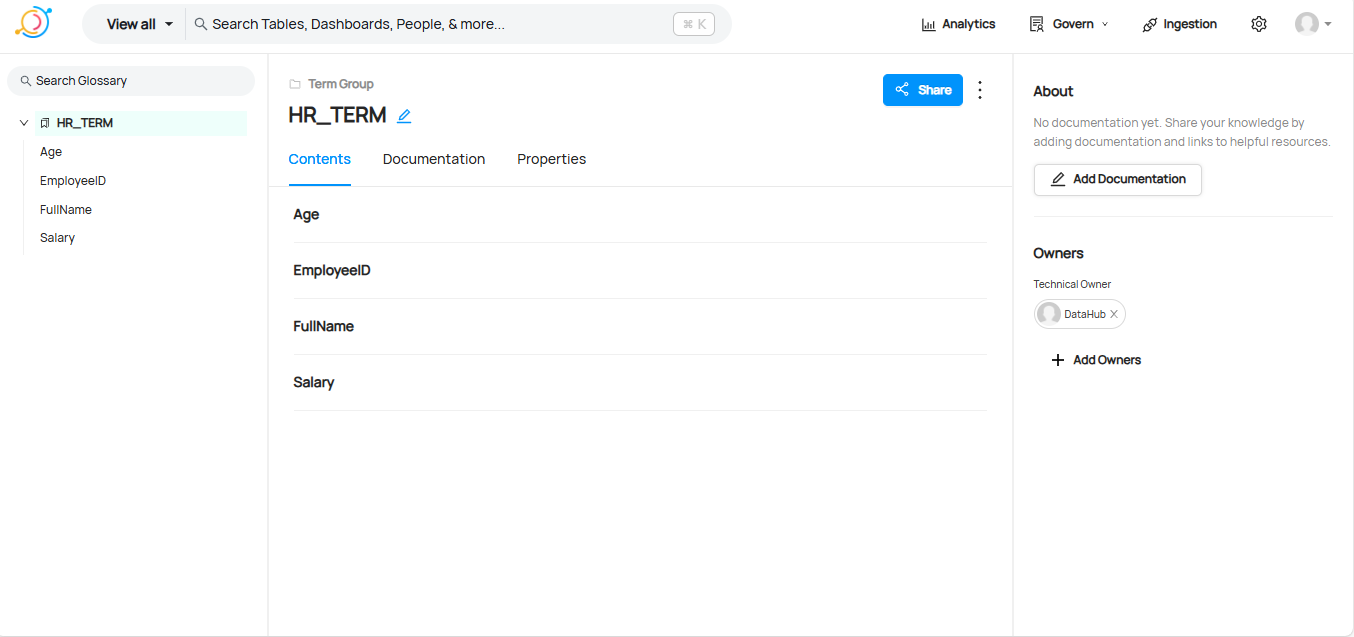


Figure II. 7 Glossary management interface for HR term group

The image displays DataHub's glossary management interface for the "HR\_TERM" term group, a human resources data dictionary. The left sidebar shows a hierarchical glossary structure with HR\_TERM as the parent category, containing terms such as Age, EmployeeID, FullName, and Salary. The main content area, with the Contents tab selected, lists these HR-related terms, with additional tabs for Documentation and Properties available. The right panel indicates ownership by "DataHub" as the Technical Owner, with options to add documentation and additional owners. This interface highlights DataHub's ability to organize and manage business terminology in a structured glossary, supporting data governance and consistent understanding of HR data elements across the organization.