Politecnico di Torino

Politecnico di Torino

Master's Degree in Data Science and Engineering

A.a. 2025/2026

Graduation Session March 2026

Design and Implementation of a Metadata-Driven Enterprise ETL Framework on Databricks

Bridging Data Engineering and AI through Automated Pipelines and Governed Data Assets

Academic Supervisor:
Prof. Daniele Apiletti

Company Supervisor:
Dr. Filippo Balla

Candidate:
Aurora Leone

Abstract

Modern enterprises require unified platforms that integrate data engineering, governance, and analytics to support strategic decision-making. This thesis evaluates Databricks as a cloud-native environment for implementing and extending a structured, metadata-driven ETL framework developed by Data Science Operations.

The framework follows a modular three-layer architecture: raw data ingestion (L0), integration and quality control (L1), and analytical publication (L2). It was fully re-engineered on Databricks using Delta Lake for transactional consistency and Databricks Jobs for orchestration. Metadata-driven automation, dependency-aware scheduling, and comprehensive auditing mechanisms ensure traceability, reproducibility, and operational efficiency across the entire pipeline.

Beyond traditional ETL operations, the study demonstrates integration with advanced analytics workflows. A machine learning model was developed to estimate the probability of credit risk deterioration, utilizing curated datasets produced by the framework. The model incorporates monotonic constraints and probability calibration to ensure financial interpretability and business alignment. This process simulates organizational role separation between data engineering and data science teams, leveraging Unity Catalog's governance layer to enforce fine-grained access controls. Time series forecasting functionalities were also explored to evaluate their applicability to financial scenarios, providing a critical assessment of platform capabilities despite dataset limitations.

To illustrate business value, a visualization dashboard was created to transform engineered data assets into actionable insights for stakeholders. The findings demonstrate that Databricks provides a cohesive, scalable environment capable of unifying data engineering, governance, and predictive analytics, offering a robust alternative to fragmented traditional architectures.

Table of Contents

List of Tables

List of Figures

Listings

Chapter 1

Introduction

1.1 Context and Motivation

In the contemporary digital economy, data has become a strategic asset for organizations operating across all sectors. The continuous growth in data volume, velocity, and variety — the three defining dimensions of the Big Data phenomenon first systematically characterized by Laney [1] — has transformed the way enterprises design their information systems and decision-making processes. Modern organizations no longer rely solely on transactional systems for operational activities; instead, they require scalable platforms capable of integrating heterogeneous data sources, ensuring quality and governance, and enabling advanced analytics and artificial intelligence applications.

Cloud-native analytics platforms have emerged as a response to these evolving needs [2]. Compared to traditional on-premise solutions, cloud environments offer elastic scalability, distributed processing, reduced infrastructure maintenance, and integrated services for data engineering, analytics, and machine learning. Among these platforms, Databricks has gained significant attention due to its unified approach that combines data engineering, data warehousing, and data science within a single ecosystem built on Apache Spark [3] and Delta Lake [4] technologies.

Data Science Operations, a company specialized in data integration and digital transformation projects, has historically developed and maintained a proprietary ETL framework to manage structured and automated data pipelines for enterprise clients. As the technological landscape progressively shifts toward cloud-native architectures, the company recognized the need to evaluate modern platforms capable of hosting, modernizing, and potentially enhancing its existing framework. In this context, Databricks was selected as a candidate environment to assess its suitability for implementing the corporate ETL architecture while extending its capabilities to support advanced analytics and machine learning use cases.

1.2 Problem Statement

The adoption of cloud-native platforms for enterprise data management raises critical questions regarding architectural compatibility and functional completeness. Specifically, it remains unclear whether platforms such as Databricks can effectively support structured, metadata-driven ETL frameworks — whose design principles are rooted in the data warehouse tradition [5] — while preserving governance, automation, and scalability requirements within a modern lakehouse environment [6].

This thesis addresses two interconnected challenges:

1. **ETL Framework Migration:** Can the Data Science Operations three-layer ETL architecture be successfully implemented on Databricks while maintaining its core principles of metadata governance, traceability, and quality control?

2. **Analytical Integration:** Can the same platform seamlessly support downstream analytics and machine learning workflows, providing an end-to-end solution from data ingestion to AI-powered insights?

Answering these questions requires demonstrating technical feasibility, evaluating performance, and assessing the platform's ability to unify traditionally separate data engineering and data science functions.

1.3 Objectives

The primary ambition of this thesis is to demonstrate the feasibility and strategic advantages of migrating a structured enterprise data architecture to a unified cloud-native environment. To this end, the research pursues a set of interconnected objectives spanning foundational data engineering practices and advanced analytical applications.

The **main objective** of this thesis consists in the implementation of the Data Science Operations ETL framework on the Databricks platform up to the Integration layer (L1). Specifically, the work reproduces the first two layers of the framework's three-tier architecture — Ingestion (L0) and Integration (L1) — through the development of staging and Operational Data Store (ODS) tables. This implementation leverages the distributed processing capabilities of PySpark and the transactional reliability provided by Delta Lake [4].

Particular emphasis is placed on the design and orchestration of structured data pipelines to ensure data quality, consistency, and traceability across layers. Although the Semantic layer (L2) is not implemented within the scope of this thesis, the proposed architectural design and data modeling choices are conceived to enable its future extension toward analytical consumption and business reporting.

Beyond the core architectural implementation, the thesis addresses several **secondary objectives** aimed at exploring the broader capabilities of the Data Lakehouse paradigm [6].

First, the study evaluates governance and auditing mechanisms by implementing metadata-driven control tables and simulating distinct organizational roles. This approach enables an assessment of data lineage tracking, access control enforcement, and security management throughout the data lifecycle within the Databricks environment.

Second, to demonstrate the platform's value for downstream analytical use cases, a machine learning model for credit risk prediction is developed [7]. This objective is particularly relevant as it illustrates how curated data produced by the ETL process can be seamlessly operationalized into predictive models within a regulated financial context [8], while preserving a clear separation of duties between Data Engineering and Data Science functions.

Furthermore, the research includes an exploratory component focused on time series forecasting using Databricks AutoML, which evaluates additive decomposition models [9] alongside classical statistical and deep learning alternatives. Despite inherent dataset limitations, this analysis serves to evaluate the technical integration, scalability, and practical applicability of the platform's native forecasting capabilities within a financial context.

Finally, the work culminates in the development of a business-oriented dashboard. This visualization layer bridges the gap between technical data engineering processes and strategic decision-making, demonstrating how engineered data assets can be translated into actionable insights for stakeholders.

Collectively, these objectives aim to assess whether Databricks can effectively function as a unified platform for enterprise-grade data engineering, governance, and advanced analytics.

1.4 Methodology

The methodology adopted in this work follows an implementation-based approach with exploratory components.

Initially, an in-depth analysis of the Databricks platform was conducted to understand its architectural components, governance mechanisms, and storage model. Based on this analysis, the company ETL framework was adapted to the Databricks environment, preserving its logical structure while leveraging platform-native features such as Delta Lake [4] and Auto Loader [10].

The implementation was structured into modular pipelines where Python and PySpark [3] were used for ingestion, validation, and integration tasks. Orchestration was configured through Databricks Jobs [11] to manage dependencies and automate

execution sequences.

Subsequently, the curated datasets were used to develop two distinct analytical applications: a machine learning model for credit risk assessment [7] and a series of time series forecasting experiments [9]. Governance aspects were explicitly tested by restricting access to selected datasets, simulating real organizational data-sharing constraints. The final stage involved implementing a visualization layer to assess the communicative effectiveness of the platform for business stakeholders.

1.5 Structure of the Thesis

The remainder of this thesis is organized as follows.

Chapter 2 presents the state of the art, introducing data management systems, data warehouses, data lakes, and modern cloud-native analytics platforms.

Chapter 3 describes the Data Science Operations ETL framework, including its architectural design, layering strategy, and metadata governance model.

Chapter 4 details the implementation of the framework on Databricks, covering ingestion mechanisms, pipeline orchestration, auditing, and monitoring.

Chapter 5 illustrates the artificial intelligence applications, focusing on the credit risk prediction model and time series forecasting experiments.

Chapter 6 presents the empirical results of both the ETL implementation and the analytical models, covering execution performance, pipeline scalability and reusability, monitoring and operational visibility, machine learning model performance and risk segmentation, time series forecasting outcomes, and business-oriented visualization.

Chapter 7 provides a critical discussion of the proposed solution, examining its architectural generalizability, key strengths — including deterministic change detection, governance by construction, and separation of concerns — alongside its current limitations and a comparison with traditional ETL approaches.

Finally, Chapter 8 summarizes the main contributions of the thesis and outlines future development directions, including ingestion automation, completion of the L2 layer and MDM enrichment, advanced orchestration, MLOps integration, cost monitoring, and extension to streaming workloads.

Chapter 2

State of the Art

2.1 Data Management Systems

Data Management Systems represent the technological foundation through which organizations collect, store, process, and govern data across its entire lifecycle. As data volumes, variety, and velocity have grown [1], these systems have evolved to address heterogeneous requirements ranging from real-time transaction handling to large-scale analytical processing. The design of a data management architecture is strongly influenced by the nature of the workload it must support, as well as by constraints related to performance, consistency, scalability, and data governance.

In enterprise environments, data management systems are not monolithic but rather composed of multiple specialized components, each optimized for a specific function within the broader information ecosystem. Operational systems focus on supporting business processes with strict latency and consistency requirements, while analytical systems are designed to enable data exploration, reporting, and advanced analytics on historical datasets. This functional separation has historically driven the emergence of distinct architectural paradigms — most notably Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) systems [12] — each tailored to a specific usage pattern. Understanding their characteristics and limitations is essential to motivate the architectural choices discussed in the following sections, as well as the evolution toward more integrated solutions such as Data Warehouses [13], Data Lakes [14], and modern Lakehouse platforms [15].

2.1.1 OLTP and OLAP Systems

OLTP and OLAP systems are optimized for fundamentally different technical requirements, reflecting distinct operational objectives within the enterprise data ecosystem [12].

OLTP systems are engineered to support daily business operations by handling a large number of concurrent, short-lived transactions — inserts, updates, and deletes. Typical examples include banking transaction systems, e-commerce platforms, and real-time inventory management applications. These systems are built around the **ACID** properties — Atomicity, Consistency, Isolation, and Durability — which collectively guarantee that each transaction is processed reliably and that the database remains consistent even in the presence of concurrent access or system failures [16]. To minimize redundancy and optimize write performance, OLTP databases rely on highly normalized relational schemas, often conforming to the Third Normal Form (3NF). While this makes them well suited for point queries and transactional workloads, it also makes them poorly suited to complex analytical processing.

OLAP systems, by contrast, are designed to support data analysis, reporting, and strategic decision-making [17]. They consolidate large volumes of historical data from multiple operational sources to enable trend analysis, pattern discovery, and performance monitoring. OLAP workloads are predominantly read-intensive and involve long-running queries performing multi-dimensional aggregations — such as evaluating the average probability of default across an entire financial portfolio. To improve query performance, OLAP environments typically adopt denormalized data models, including Star or Snowflake schemas [5], which minimize join complexity and accelerate aggregation operations.

A fundamental distinction between the two paradigms lies in their temporal focus. OLTP systems reflect the current operational state of the business, while OLAP systems provide a longitudinal view over historical data. Running complex analytical queries directly on transactional systems would cause significant resource contention, risking degradation of mission-critical operations. This incompatibility has historically motivated the adoption of dedicated data integration pipelines to transfer and reshape data from operational systems into analytical platforms.

2.1.2 ETL and ELT Integration Pipelines

Two principal approaches have emerged to bridge the gap between operational and analytical environments. **Extract, Transform, Load (ETL)** pipelines extract data from source systems, apply transformations and cleansing in a dedicated staging environment, and load the resulting structured data into the target platform. **Extract, Load, Transform (ELT)** pipelines instead load raw data directly into the target system and defer transformation to the analytical layer, exploiting the processing power of modern distributed engines [5].

While ETL favors enforcing data quality before ingestion, ELT prioritizes ingestion speed and flexibility — making it particularly well suited to cloud-native and lakehouse architectures where compute and storage are decoupled [6]. Both

paradigms preserve the architectural separation between operational and analytical environments, though at the cost of introducing latency, potential data duplication, and increased maintenance complexity — limitations that modern architectures such as the Data Lakehouse aim to address.

2.1.3 Data Warehouse Concepts and Architectures

Building upon the separation between operational and analytical workloads, the data warehouse represents the canonical architectural solution for enterprise-scale analytical processing [18]. It provides a structured, reliable, and scalable environment designed to support reporting, business intelligence, and advanced analytics.

A data warehouse is a centralized repository optimized for analytical rather than transactional workloads. It integrates data from multiple heterogeneous operational sources and retains both current and historical information, allowing organizations to analyze business performance over extended time horizons. Inmon [18] characterized the data warehouse as "a subject-oriented, integrated, time-variant, and non-volatile collection of data in support of management decision-making" — properties that distinguish it from operational databases by emphasizing long-term retention, stability, and analytical usability.

From an architectural standpoint, a data warehouse is supported by interconnected components governing data ingestion, transformation, storage, access, and governance. Integration is typically implemented through ETL or ELT pipelines operating in batch or near-real-time modes, though hybrid approaches combining data virtualization and physical storage are also common where latency and operational constraints require it.

Metadata management plays a central role within these architectures. Technical metadata captures schemas, transformation logic, and data lineage, while business metadata provides semantic definitions and contextual meaning. Together, they support trust in analytical results and underpin governance and regulatory compliance. Security policies, access control, data quality enforcement, and lineage tracking complete the governance layer, ensuring that analytical activities remain compliant throughout the data lifecycle.

Two design methodologies have most shaped the evolution of data warehouse architectures [5, 18]. The Inmon approach follows a top-down strategy centered on a normalized enterprise data warehouse serving as a single source of truth, prioritizing integration and governance from the outset. The Kimball approach adopts a bottom-up methodology based on dimensional data marts organized around specific business processes, emphasizing usability, query performance, and rapid delivery of analytical value. In practice, contemporary enterprise environments frequently combine both, pairing centralized governance with dimensional models optimized for consumption.

2.1.4 Dimensional Data Modeling

Within data warehouse architectures, dimensional data modeling represents one of the most widely adopted techniques for structuring data to support analytical workloads [5]. Closely associated with the Kimball methodology, it provides a logical design approach that aligns technical data structures with business-oriented analysis requirements.

At the core of dimensional modeling is the distinction between fact tables and dimension tables. Fact tables capture quantitative measurements associated with business events — such as transaction amounts, balances, or counts — and are typically additive or semi-additive in nature. Dimension tables provide the descriptive context required to interpret these measures, representing entities such as time, customer, product, or organizational structure. This separation enables analytical queries to aggregate metrics along multiple business perspectives efficiently.

Dimension tables often incorporate hierarchical relationships that support analysis at varying levels of granularity. For example, a time dimension may support drill-down from annual totals to monthly or daily figures, while a customer dimension may enable aggregation from individual accounts to segments or regions. Dimensional modeling also accommodates historical changes through the concept of Slowly Changing Dimensions [5], a set of techniques determining how attribute changes in source systems are tracked and preserved in the warehouse.

By favoring simplified and largely denormalized structures, dimensional models are optimized for analytical query performance and ease of use. Three principal schema variants are commonly employed, each representing a different trade-off between simplicity, redundancy, and structural complexity.

Star Schema

The star schema is the most straightforward and widely used implementation of dimensional modeling [5]. It consists of a central fact table directly linked to a set of fully denormalized dimension tables. The key advantage of this structure is that it minimizes join complexity: a query aggregating monthly sales by product category and region requires joining only the central fact table with three dimension tables, resulting in highly efficient execution. The principal trade-off lies in increased data redundancy within dimension tables — for instance, a product category name may be repeated across thousands of rows — which is generally accepted in exchange for improved performance and simplicity.

Snowflake Schema

The snowflake schema is a variation of the star schema in which one or more dimensions are further normalized into multiple related tables [5]. Consider a customer dimension: rather than storing country, region, and city directly in a single table, the snowflake schema separates them into a hierarchy of linked tables. This design reduces redundancy and enhances referential integrity, particularly when dealing with complex hierarchical relationships. However, the additional joins introduced by normalization can increase query complexity and may negatively affect performance compared to a star schema. Snowflake schemas are therefore typically adopted when maintainability and consistency outweigh performance considerations.

Galaxy Schema

The galaxy schema, also referred to as a fact constellation, extends dimensional modeling to support multiple fact tables that share common dimension tables [5]. In a financial institution, for example, a galaxy schema might combine a loan origination fact table with a payment transaction fact table, both sharing common dimensions such as customer, date, and branch. This approach enables integrated analysis across different business processes — such as combining sales, finance, and risk metrics within a single analytical environment. Compared to the star and snowflake variants, galaxy schemas require more careful design to manage increased structural and query complexity, but offer greater analytical breadth across heterogeneous business domains.

2.1.5 Data Marts and Analytical Use Cases

Within enterprise data warehouse environments, data marts represent a logical and organizational abstraction designed to address the analytical requirements of specific business domains or organizational units [5]. Rather than serving as standalone systems, data marts are typically derived from the centralized analytical repository and provide focused, curated views tailored to well-defined use cases, such as finance, risk management, or sales analysis.

Data marts are characterized by a limited and well-scoped set of measures and dimensions, which simplifies data consumption and enhances usability for domain experts. By exposing only the data relevant to a specific context, they reduce cognitive and technical complexity while enabling faster query execution and more intuitive analytical workflows. Intermediate integration layers — such as Operational Data Stores or curated analytical zones — are frequently employed to ensure consistency, cleansing, and semantic alignment prior to data mart construction. From a governance perspective, data marts must remain consistent

with enterprise-wide data definitions and quality standards, enabling decentralized analytical access while preserving centralized control over lineage, security, and compliance.

As data volumes and analytical diversity grow, however, the data mart model alone proves insufficient. The increasing need to store heterogeneous, raw, and semi-structured data at scale motivated the emergence of the data lake paradigm, discussed in the following section.

2.2 Data Lakes and Modern Architectures

Building on the architectural principles of data warehouses, data lakes emerged as a complementary paradigm aimed at addressing the growing diversity, scale, and dynamism of modern data [19, 14]. While data warehouses provide structured, governed environments optimized for analytical consistency, data lakes prioritize flexibility and scalability, enabling organizations to retain and exploit data whose analytical value may not be immediately apparent at the time of ingestion.

2.2.1 Definition and Key Characteristics of Data Lakes

Modern organizations must manage not only structured transactional data, but also semi-structured and unstructured sources such as application logs, sensor data, multimedia content, and external data streams [20]. These data types challenge traditional data warehouse approaches, which rely on predefined schemas and tightly controlled ingestion pipelines.

A data lake is a centralized repository that stores data in its raw, native format until it is required for analysis [19]. Unlike data warehouses, which adopt a *schema-on-write* approach — enforcing data modeling and transformation prior to ingestion — data lakes rely on a *schema-on-read* paradigm, in which structure and semantics are applied at query time [21]. This deferred interpretation reduces the upfront cost of data onboarding and allows greater adaptability to evolving analytical requirements.

By leveraging distributed storage systems — most commonly cloud-based object storage — data lakes decouple storage from compute, enabling independent and cost-efficient scaling [2]. This design supports both batch and streaming ingestion from heterogeneous sources and makes data lakes particularly well suited to machine learning and advanced analytics workloads, where raw data preservation allows analysts to explore datasets iteratively, apply alternative transformations, and test diverse approaches without being constrained by rigid schemas.

The flexibility of data lakes, however, introduces significant governance challenges. Without adequate metadata management, cataloging, and quality controls, data lakes risk becoming so-called *data swamps* — repositories where data accumulates

without sufficient structure or documentation, becoming effectively undiscoverable and untrustworthy [21]. These aspects are addressed in depth in Section 2.3.

2.2.2 Comparison Between Data Warehouse and Data Lake

Data warehouses and data lakes represent distinct yet complementary approaches to data management [18, 19]. Data warehouses provide curated, structured repositories with schema-on-write enforcement, delivering consistency, quality, and predictable query performance suited to standardized reporting and business intelligence. Data lakes, by contrast, accept raw data with minimal upfront processing, supporting a broader range of formats and workloads — particularly in big data analytics and machine learning — at the cost of weaker native governance [21].

Rather than mutually exclusive alternatives, the two paradigms increasingly coexist within modern architectures, each serving distinct but complementary roles. Their respective strengths and limitations have motivated the development of unified solutions that combine the governance and performance of data warehouses with the openness and scalability of data lakes — a convergence that directly inspired the Lakehouse architecture [6, 15].

2.2.3 The Lakehouse Paradigm

The limitations of both traditional data warehouses and early data lake implementations led many organizations to adopt hybrid two-tier architectures, in which raw data is first ingested into a data lake and subsequently transformed and replicated into a data warehouse for analytical consumption. While effective in separating exploratory and structured workloads, this approach introduces data duplication, increased latency, and higher operational complexity — challenges that are particularly acute in machine learning and advanced analytics scenarios, where large-scale and timely access to data is essential.

In response to these shortcomings, the Lakehouse paradigm has emerged as a unifying architectural model that combines the reliability and performance of data warehouses with the scalability and openness of data lakes [6, 15]. A lakehouse is typically built on cloud-based object storage using open columnar file formats such as Apache Parquet or ORC, preserving the cost efficiency and interoperability of data lakes while introducing data warehouse capabilities such as ACID transactions, schema enforcement, data versioning, and query optimization [4].

These capabilities are enabled through a centralized transactional metadata layer that manages table definitions, data consistency, and lineage information. This metadata-driven approach supports concurrent workloads and incremental updates while maintaining governance and reliability, effectively eliminating the risk of data swamps by imposing structure and governance controls without sacrificing

the capacity to store and process raw or semi-structured data.

A defining characteristic of the lakehouse is its ability to support multiple analytical workloads within a single platform. Business intelligence, ad hoc analytics, machine learning, and data science applications can operate directly on the same datasets, eliminating the need for costly data movement across systems [6]. In contemporary enterprise environments, data warehouses, data lakes, and lakehouses often coexist within a broader data ecosystem, with the lakehouse increasingly acting as a convergence layer that unifies analytics and AI workloads on a shared data foundation.

2.3 Metadata Management and Data Governance

As data architectures evolve from traditional data warehouses to modern lakehouse platforms, metadata management has become a foundational enabler for data quality, governance, and usability [22]. In environments characterized by heterogeneous sources, flexible ingestion patterns, and schema-on-read paradigms, metadata provides the contextual layer necessary to understand, interpret, and trust data assets across the entire analytical stack.

2.3.1 Role of Metadata in Data Platforms

Contemporary data-driven organizations operate across complex ecosystems spanning on-premises databases, cloud object storage, data lakes, and hybrid lakehouse platforms. These architectures introduce fragmentation, data silos, and governance challenges, further intensified by regulatory frameworks such as the General Data Protection Regulation (GDPR) [23], the California Consumer Privacy Act (CCPA) [24], and the Health Insurance Portability and Accountability Act (HIPAA) [25].

Within this context, metadata functions as the connective tissue linking disparate systems and datasets [22]. It encompasses three complementary categories. *Technical metadata* captures structural information: schemas, data types, transformation logic, and lineage paths. *Operational metadata* records execution context: pipeline run times, record counts, error logs, and processing durations. *Business metadata* provides semantic meaning: glossary definitions, ownership assignments, and regulatory classifications. Together, these layers allow all stakeholders — data engineers, analysts, compliance officers, and business users — to locate, understand, and evaluate data assets irrespective of their physical location or format.

Modern data catalogs [26] enhance traditional metadata management through automation and machine learning. Automated classification can identify data types, infer schemas, and tag sensitive information, while anomaly detection and

natural language processing further improve discoverability and semantic alignment between technical assets and business glossaries.

2.3.2 Data Lineage and Cataloging

A primary application of metadata management is enabling robust data governance through centralized data catalogs [22]. These repositories provide a searchable inventory of enterprise data assets, allowing users to discover datasets, inspect their structure and semantics, and evaluate their suitability for specific analytical tasks.

Beyond discovery, metadata captures data provenance — the origin, transformation history, and movement of data across systems — collectively referred to as data lineage [27]. Lineage information ensures data integrity, facilitates troubleshooting of pipeline failures, and provides auditability. In regulated contexts, lineage is crucial for demonstrating compliance by documenting how sensitive data is collected, transformed, and consumed [23]. Modern data catalogs embed governance mechanisms directly, incorporating business glossaries, stewardship assignments, access policies, and regulatory constraints, thus transforming metadata from a passive reference layer into an active governance tool.

2.3.3 Data Quality, Auditing, and Compliance

Effective data quality management relies on the systematic definition and enforcement of quality rules grounded in metadata attributes. Completeness checks verify that required fields are populated; consistency rules detect mismatches across integrated sources; timeliness controls flag records that fall outside expected ingestion windows. These rules can be applied automatically at each pipeline stage, enabling anomalies and violations to be intercepted before data reaches analytical layers.

Auditing capabilities leverage lineage metadata to provide end-to-end traceability from source systems to dashboards or reports [27]. This traceability is essential for both operational oversight and regulatory requirements: under frameworks such as GDPR [23] and HIPAA [25], organizations must be able to demonstrate where personal or sensitive data originates, how it is transformed, and who has accessed it. Metadata-driven auditing enables the identification of errors, assessment of pipeline performance, and documentation of corrective actions, thereby strengthening accountability and trust across the entire data ecosystem.

Taken together, the governance mechanisms described in this section — catalogs, lineage, quality enforcement, and compliance auditing — form the foundational layer upon which modern analytics platforms must operate. The following section examines how these principles are realized in practice within Databricks [28] as a representative implementation of the lakehouse architecture.

2.4 Modern Analytics Platforms

The evolution from traditional data warehouses to data lakes and lakehouse architectures has transformed analytics platforms into unified environments capable of supporting the entire data lifecycle [6, 15]. Modern platforms are expected to integrate data ingestion, transformation, governance, analytics, and machine learning within a single scalable framework, while remaining adaptable to changing business requirements and regulatory constraints. Cloud-native architectures have become central to this evolution [2]: by decoupling storage and compute, leveraging elastic resources, and adopting managed services, they reduce operational complexity and enable collaboration across multidisciplinary teams. Databricks [28] represents one of the most prominent implementations of this class of platforms, and its architecture is examined in detail in the following subsections.

2.4.1 Databricks Platform: Vision and Architecture

Databricks is a cloud-native analytics platform developed to address the fragmentation and operational complexity typical of earlier big data ecosystems. Founded in 2013 by the original creators of Apache Spark at the University of California, Berkeley [3], the platform combines strong academic foundations with enterprise-grade requirements. Its core objective is to simplify large-scale data analytics and machine learning by abstracting infrastructure complexity and unifying diverse analytical workflows within a cohesive, managed Software-as-a-Service (SaaS) environment [28].

A defining feature of Databricks is its native integration with Apache Spark [3], which serves as the distributed processing engine. Spark provides a unified execution model for batch processing, streaming analytics, interactive queries, and machine learning, and Databricks enhances it with cloud-native optimizations and tight integration with storage and governance layers, while remaining aligned with the open-source ecosystem.

The platform's architecture is built around the Lakehouse paradigm described in Section 2.2.3, combining the scalability and openness of data lakes with the reliability and governance traditionally associated with data warehouses [6]. At the storage layer, Databricks leverages cloud object storage services such as Amazon S3, Azure Data Lake Storage, and Google Cloud Storage, storing data in open formats to ensure interoperability and avoid vendor lock-in. Delta Lake [4], an open-source storage framework built on top of this layer, adds ACID transactions, schema enforcement, time travel, and versioning capabilities — its architecture is described in greater detail in Section 2.4.2.

Compute resources are fully decoupled from storage and dynamically provisioned in the cloud [2]. Managed Spark clusters can be tailored to workloads ranging from

interactive analytics and batch jobs to real-time streaming and large-scale machine learning training. Automated optimizations, including adaptive query execution and intelligent caching, reduce operational overhead while improving performance and cost efficiency.

Collaboration is facilitated through a unified notebook-based workspace supporting Python, SQL, Scala, and R. This environment integrates code, documentation, visualizations, and results, enhancing reproducibility, transparency, and cross-team collaboration across data engineers, data scientists, and analysts. Databricks operates on a multi-cloud strategy, available on AWS, Microsoft Azure, and Google Cloud Platform, with deep native integrations on each.

2.4.2 Delta Lake and Transactional Data Management

At the core of the Databricks Lakehouse is Delta Lake [4], an open-source storage framework that introduces transactional guarantees and structured data management to cloud object storage. Its primary contribution is the extension of ACID transaction semantics [16] to a storage layer that was originally designed for unstructured, append-only operations, thereby addressing the consistency, corruption, and governance weaknesses that characterized early data lake implementations [21].

The mechanism underpinning these guarantees is a centralized, append-only transaction log that records every operation applied to a table — including inserts, updates, deletes, and schema changes. This log enables concurrent reads and writes through optimistic concurrency control, supports rollback to any prior table version, and provides a complete and queryable audit history of all data modifications [4]. Schema enforcement prevents the ingestion of malformed or incompatible records at write time, while a controlled schema evolution mechanism allows table definitions to be updated over time as analytical requirements change. Time travel capabilities allow users to query any historical snapshot of a table by specifying a version number or timestamp, directly supporting analytical reproducibility and regulatory auditing requirements [23].

Delta Lake integrates natively with the ELT workflows common in lakehouse architectures [6], enabling data to be loaded incrementally and transformed in place without duplication. This design ensures that raw and curated data can coexist within the same storage layer, providing a unified foundation for diverse workloads including reporting, machine learning, and real-time analytics.

2.4.3 Machine Learning and Advanced Analytics Capabilities

Databricks provides native support for the end-to-end machine learning lifecycle through MLflow [29], an open-source platform that manages experiment tracking,

model versioning, and deployment. MLflow enables data scientists to log parameters, metrics, and artifacts for each training run, compare experiments systematically, and register production-ready models in a governed model registry. This integration bridges the gap between exploratory data science and production machine learning, enabling reproducible, governed, and scalable model development.

Beyond MLflow, Databricks offers AutoML — a glass-box automation layer that generates baseline model code for classification, regression, and time series forecasting tasks. Unlike black-box AutoML tools, Databricks AutoML exposes the generated notebooks, allowing practitioners to inspect, extend, and customize the modeling pipeline. This transparency is particularly valuable in regulated domains such as finance and healthcare, where model explainability and auditability are mandatory requirements.

The platform also supports feature engineering at scale through the Feature Store, a centralized repository that enables features to be computed once and reused across multiple models, preventing training-serving skew and promoting consistency between offline training and online inference. Distributed training frameworks — including Horovod for deep learning and built-in hyperparameter tuning — allow large-scale model training to be parallelized across GPU and CPU clusters, making Databricks well suited to both classical machine learning and deep learning workloads. These capabilities are directly leveraged in this thesis for credit risk prediction [7] and time series forecasting [9], as detailed in Chapter 5.

2.4.4 Security, Governance, and Access Control

A key component of Databricks' governance framework is *Unity Catalog* [30], which provides centralized management of metadata, access policies, and auditing across the entire lakehouse. Unity Catalog enables fine-grained access control at the table, row, and column levels, ensuring that sensitive data is appropriately protected while remaining accessible to authorized users. It implements the governance principles described in Section 2.3 directly within the analytical platform, bridging the gap between policy definition and operational enforcement [6].

Databricks also supports secure data sharing through *Delta Sharing* [31, 4], an open protocol that enables real-time, governed data exchange across organizations and platforms without requiring data replication or vendor-specific tooling. When combined with Unity Catalog, Delta Sharing ensures that shared datasets remain compliant, auditable, and consistent with organizational security policies [23], extending governance beyond the boundaries of a single enterprise environment.

2.4.5 Enterprise Adoption and Use Cases

Databricks has achieved widespread adoption across industries as a foundational platform for advanced analytics and artificial intelligence, driven by its ability to unify data engineering, machine learning, and governance within a single coherent environment [6, 28].

In the financial services sector, institutions leverage the lakehouse architecture to consolidate data from disparate operational systems — core banking, payment processors, market data feeds — enabling real-time fraud detection, credit risk modeling [7], regulatory reporting, and customer segmentation at scale. The combination of Delta Lake's transactional guarantees with Spark's distributed processing makes it possible to run complex risk models over large historical datasets while simultaneously ingesting live transaction streams.

In retail and e-commerce, Databricks supports demand forecasting, real-time personalization engines, and supply chain optimization. Healthcare and life sciences organizations use the platform for genomic data analysis, clinical trial management, patient outcome prediction, and pharmacovigilance. In manufacturing, predictive maintenance use cases benefit from the platform's ability to ingest high-frequency sensor data from industrial equipment, train failure prediction models, and deploy them at the edge or in near-real-time pipelines. Across all these sectors, the platform's scalability, openness, and native MLflow integration [29] enable organizations to move from data ingestion to production model deployment within a unified framework.

2.4.6 Comparison with Alternative Platforms

Databricks occupies a distinctive position in the modern data platform landscape by offering a unified, cloud-native environment that spans data engineering, transformation, governance, analytics, and machine learning within a single architecture [6]. Built around Apache Spark [3] and the lakehouse paradigm, it supports structured and unstructured data, real-time streaming, large-scale feature engineering, and end-to-end AI pipelines — making it particularly well suited to organizations with complex, heterogeneous, and rapidly evolving analytical requirements.

Its primary competitors adopt different architectural philosophies. Snowflake [32] is optimized for structured analytical workloads, offering exceptional query performance, a fully managed multi-cluster architecture, and strong data sharing capabilities through its separation of storage and compute. Its SQL-first interface makes it highly accessible for teams focused on business intelligence and standardized reporting, though it offers significantly less flexibility for advanced machine learning workflows and custom data engineering compared to Databricks.

Google BigQuery, built on the Dremel distributed query engine [33], follows a serverless model with automatic scaling and deep integration with the Google

Cloud ecosystem. It excels at large-scale SQL analytics without infrastructure management, and its integration with Vertex AI provides a pathway to machine learning. However, its architecture is less suited to iterative, notebook-driven data science workflows or complex streaming pipelines that require fine-grained control.

Microsoft Fabric [34] represents a more recent entrant that consolidates data engineering, warehousing, and business intelligence within the Microsoft ecosystem, with strong integration with Power BI and Azure services. Its low-code analytics capabilities and unified SaaS model lower the barrier to entry for organizations already invested in the Microsoft stack, though at the cost of reduced flexibility for highly customized data engineering or advanced AI workflows.

Despite its strengths, Databricks is not universally optimal. Its depth and flexibility require skilled engineering teams capable of managing clusters, optimizing pipelines, and controlling costs — operational overhead that may not be justified for organizations whose primary workload consists of standardized reporting or simple data consolidation. In such contexts, simpler platforms with lower operational complexity may deliver greater value.

In practice, enterprises increasingly adopt hybrid architectures that combine Databricks for complex data engineering and machine learning with complementary platforms for governance, reporting, or business-user consumption. This reflects a broader trend in modern data ecosystems: no single platform satisfies all analytical requirements across an organization, and strategic value is maximized by integrating complementary strengths within a coherent, governed data architecture [6] rather than seeking a universal solution.

Chapter 3

Data Science Operations ETL Framework

This chapter presents the ETL framework developed by Data Science Operations (DSO), which constitutes the methodological foundation upon which the implementation described in this thesis is built. As discussed in Chapter 2, the evolution toward cloud-native Lakehouse architectures [6] has created the need for structured, governance-oriented integration frameworks capable of operating natively within distributed environments such as Databricks [28]. The DSO framework was designed precisely to meet these requirements, providing a modular, metadata-driven architecture for enterprise data integration that handles heterogeneous source systems and supports advanced analytical workloads.

The framework is organized into three logical processing layers — L0, L1, and L2 — each responsible for a distinct stage of the data lifecycle, from raw ingestion to analytical publication. A cross-cutting metadata governance layer orchestrates execution, auditing, and dynamic configuration across all stages. Figure 3.1 provides a high-level overview of this architecture. The following sections describe the organizational context in which the framework operates, its design principles, the processing logic of each layer, the metadata governance model, and the scheduling rules that govern execution.

3.1 Organizational Context and Application Scope

Data Science Operations specializes in the design and implementation of enterprise data architectures aimed at transforming fragmented raw data into structured, high-value information assets. The company primarily serves organizations operating

in data-intensive sectors — including financial services, banking, and insurance — where regulatory compliance, auditability, and the ability to derive analytical insights from large volumes of heterogeneous operational data are critical business requirements. In these environments, the primary challenge is managing data complexity while preserving consistency, governance, and performance [22, 8].

To address these requirements, DSO developed a proprietary ETL framework that standardizes and industrializes the data integration process. Rather than constructing isolated pipelines for individual projects, the framework establishes a modular and reusable integration model that prevents the proliferation of data silos and reduces long-term maintenance overhead.

The framework is deployed in enterprise-scale environments where reliability, auditability, and scalability are essential. It consolidates heterogeneous source data into a unified analytical environment, systematically enforces validation and quality rules [22], and guarantees end-to-end traceability from source extraction to analytical publication — providing a robust foundation for both operational reporting and strategic analytics within Data Warehouse and Lakehouse architectures [6, 18].

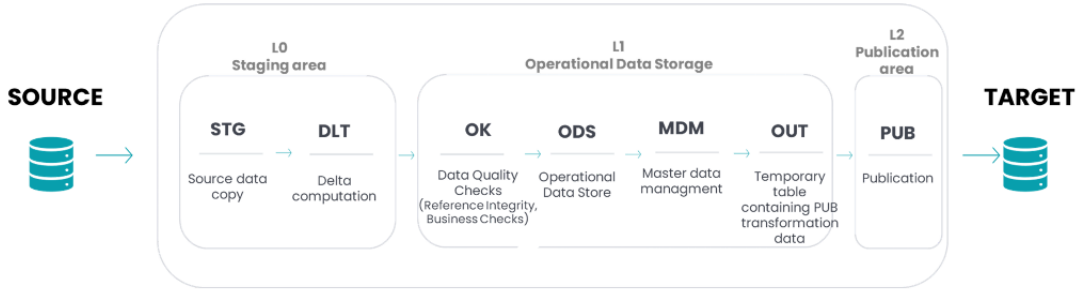

Figure 3.1: DSO ETL Framework — high-level architectural overview.

3.2 Framework Architecture and Layering Strategy

The DSO ETL framework is organized into three logical processing layers and a transversal metadata governance layer. Each processing layer fulfills a distinct role in the data lifecycle:

Layer L0 – Data Ingestion: Extracts source data and replicates it into the staging environment with minimal transformation, ensuring full traceability of the original records.

Layer L1 – Transformation and Integration: The core processing layer, where data is cleansed, normalized, integrated, validated, and enriched according to business logic and governance rules.

Layer L2 – Semantic Layer and Analytical Structuring: The final processing layer, where validated and integrated data is organized into analytical schemas optimized for Business Intelligence and advanced analytics.

Metadata Governance Layer: A cross-cutting control layer that governs execution orchestration, scheduling, auditing, and dynamic configuration across all processing stages.

This architecture embodies the principle of *Separation of Concerns* [35]: by decoupling ingestion, transformation, and publication into independent layers, the framework ensures logical independence between processing stages while maintaining coordinated execution through the metadata governance layer. This modular design improves maintainability, simplifies troubleshooting, and enables independent scalability of processing components.

A defining characteristic of the framework is its metadata-driven orchestration. Execution control, scheduling, configuration parameters, and auditing are managed through dedicated metadata tables, enabling automated monitoring and dynamic control of data flows without reliance on hard-coded logic.

3.2.1 Design Principles

The framework is grounded in five architectural principles that collectively ensure operational robustness and governance consistency.

Layered Modularity. Each layer performs a clearly defined function within the pipeline. This logical isolation prevents cross-layer dependencies, allows independent evolution of ingestion, transformation, and analytical structures, and confines the impact of changes to a single processing stage.

Traceability and Reproducibility. Every execution cycle is uniquely identified through a `JOBID` and timestamping mechanisms. This guarantees deterministic reprocessing, historical reconstruction of data states, and complete auditability of transformations across the entire pipeline [27].

Metadata-Driven Automation. Operational behavior is governed through structured metadata tables (`FLOW_MANAGER`, `TABLE_MANAGER`, and `METADATA_MANAGER`). These tables control scheduling, execution dependencies, retention rules, and monitoring parameters, enabling dynamic orchestration without modifying application code [22].

Data Quality by Design. Validation, referential integrity checks, schema enforcement, and business rule verification are embedded directly within the transformation pipeline [22]. Records failing validation are redirected to dedicated error

tables for controlled reprocessing, ensuring no information is permanently discarded without governance oversight.

Incremental Processing and Change Management. Delta-based mechanisms track insertions and deletions to minimize unnecessary data movement. This reduces computational overhead, improves performance, and ensures that only relevant changes propagate through subsequent layers.

Together, these principles align the framework with modern data engineering best practices [5, 22], ensuring governance, scalability, and operational resilience in enterprise environments.

3.3 Data Layering Strategy

The framework adopts a structured data layering strategy based on progressive semantic refinement. As data flows from L0 to L2, it evolves from a source-aligned raw representation to a fully business-aligned analytical structure. Each layer increases the level of standardization, validation, and contextual enrichment, ensuring controlled and auditable transformation at every stage of the pipeline.

3.3.1 Layer L0 – Data Ingestion

Layer L0 is the entry point of the ETL pipeline. Data is extracted from heterogeneous source systems — including operational databases, flat files, APIs, external providers, and cloud repositories — and replicated into dedicated staging structures. The objective of L0 is not to transform or validate data, but to preserve it faithfully while enabling controlled change tracking. To achieve this, the framework generates three complementary table categories: **STG** (staging) tables, **DLT** (delta) tables, and **DLT_HIS** (delta history) tables, each serving a distinct role in the ingestion and change-tracking process.

Extraction Strategies

The framework supports two extraction strategies, selected according to source system characteristics and operational requirements [5].

- **Full Extraction:** The entire dataset is retrieved from the source system. This approach ensures completeness and is applied during the initial load or when structural realignment is required. Although reliable, it may incur significant computational costs for large datasets.
- **Incremental Extraction:** Following an initial full load, subsequent runs process only new or modified records. This method reduces data movement

and execution time, and constitutes the standard approach for recurring ETL cycles.

Incremental logic relies on explicit change indicators — such as update timestamps or log tables — when available. In their absence, the framework performs snapshot comparisons between consecutive extraction cycles.

STG and DLT Table Generation

All extracted data is first stored in **STG tables**, which represent a persistent and auditable snapshot of the source at the time of ingestion. These tables contain raw records without transformation or validation. During this phase, two fundamental metadata attributes are initialized: `JOBID`, which uniquely identifies the execution run and ensures traceability across the pipeline, and `INS_TIME`, which records the precise ingestion timestamp. The STG layer guarantees reproducibility by preserving the original extracted state of the source data.

Subsequently, delta computation logic identifies changes between the current and previous snapshots. The results are stored in **DLT tables**, which capture incremental variations according to the following classification:

- Records present in the current snapshot but absent in the previous one are classified as new insertions and flagged with **FLG_NEG = 0**;
- Records present in the previous snapshot but absent in the current one are classified as logical deletions and flagged with **FLG_NEG = 1**.

This mechanism enables systematic tracking of insertions and deletions even when source systems do not provide native change data capture capabilities. Historical delta records are archived in **DLT_HIS** tables, ensuring long-term auditability without affecting the performance of active processing.

The three table categories serve complementary roles within L0. STG tables preserve the full raw dataset for auditing and recovery purposes. DLT tables isolate the incremental changes relevant for downstream processing, allowing L1 transformations to operate efficiently without reprocessing unchanged records. DLT_HIS tables archive the complete history of delta snapshots, ensuring long-term auditability without affecting the performance of active processing cycles.

3.3.2 Layer L1 – Transformation and Integration

Layer L1 is the integration and governance core of the ETL framework. Building on the raw and incremental data produced in L0, this layer applies normalization, validation, historization, enrichment, and business transformation logic to produce

datasets that are structurally consistent, semantically aligned with enterprise standards, and ready for analytical publication.

The processing flow within L1 is organized into four sequential components: OK/E$ validation, ODS consolidation, MDM enrichment, and OUT staging.

OK Layer and Error Management

The OK layer is the first formal validation checkpoint within L1. Records from the DLT tables are consolidated and harmonized according to the target schema, then subjected to a structured validation process that enforces technical and business consistency across three levels [22].

Referential integrity checks verify that all foreign key relationships are valid and that referenced entities exist in their corresponding master tables. *Schema validation* enforces non-null constraints, data type correctness, and compliance with permissible value ranges and formatting standards. *Business rule enforcement* applies domain-specific logic to detect anomalies, inconsistencies, or violations of predefined operational constraints.

Records satisfying all three validation levels are loaded into the **OK table**, which contains certified data eligible for downstream processing and is refreshed using a `TRUNCATE-INSERT` strategy to always reflect the most recent validated dataset.

Records that fail one or more validation checks are not discarded. Instead, they are redirected to the **E$ table**, the designated error repository. This table is implemented as a historized storage layer for rejected records and is managed using an `APPEND` loading strategy. This design ensures that all invalid records are persistently retained over time, thereby preserving a complete and traceable audit trail of validation failures across ingestion cycles.

A key capability of this mechanism is *error recycling*: at each new ETL execution, previously rejected records are reprocessed together with new DLT data. If prior validation issues have been resolved — for instance, through the late arrival of referenced master data — the record is promoted to the OK table and allowed to proceed through the pipeline. Otherwise, it remains in E$ for future re-evaluation. Retention policies, governed through `METADATA_MANAGER`, prevent the indefinite accumulation of irrecoverable errors while preserving auditability.

Before proceeding to the ODS, records in the OK table undergo a deduplication step to ensure that only the most recent and consistent version of each entity is propagated downstream. Each record is assigned a rank within its business primary key partition, ordered by execution recency and giving precedence to insertions over logical deletions. This ranking is computed through the following window function:

```
ROW_NUMBER() OVER (
```

```
    PARTITION BY PRIMARY_KEY
    ORDER BY JOBID DESC, FLG_NEG ASC
)
```

Only records for which this expression evaluates to `ROW_NUMBER = 1` — that is, the most recent and non-deleted version within each primary key partition — are forwarded to the ODS via a subsequent `WHERE ROW_NUMBER = 1` filter applied to the ranked result set. Records filtered at this stage are not propagated further, as their content has already been superseded within the same ingestion cycle.

Operational Data Store Layer

The Operational Data Store (ODS) is the first stable and consolidated repository of validated data within the framework. Records arriving from the OK layer are persisted using a controlled `MERGE` strategy, and system-defined primary keys are formally established to guarantee entity integrity, support referential constraints, and improve join performance through indexed access paths.

The `MERGE` operation determines, for each incoming record, whether to insert or update based on primary key existence in the target table. New records are inserted with both insertion and update metadata initialized to the current execution cycle. Existing records are updated following a Type 1 Slowly Changing Dimension (SCD Type 1) strategy [5]: descriptive attributes are overwritten with the most recent values, while the original insertion metadata remains unchanged. This approach preserves the historical trace of when each entity first entered the system while ensuring that the ODS always reflects the latest available representation of each record.

Each ODS record maintains four temporal tracking fields: `JOBID` and `INS_TIME` documenting the initial insertion, and `JOBID_UPD` and `UPD_TIME` documenting the most recent update. Together these fields provide full traceability of data evolution without requiring duplication of historical versions at this stage, since complete lineage is preserved upstream in the DLT and DLT_HIS tables.

Master Data Management Layer

The MDM layer builds upon the consolidated records in the ODS and enhances them through contextual enrichment and surrogate key governance, transitioning data from operationally consistent entities to analytically optimized master records.

The enrichment process augments ODS records with additional master and reference attributes drawn from internal repositories or external codification systems. These may include hierarchical classifications, descriptive labels, categorical groupings, and standardized external codes. Embedding this contextual information

at the MDM stage eliminates the need for runtime joins during analytical querying, simplifying downstream consumption and improving query performance at L2.

In parallel, the MDM layer introduces system-generated *surrogate keys* [5] that replace or complement natural business identifiers. Natural keys — such as customer codes or product identifiers — may be volatile, subject to redefinition, or inconsistent across heterogeneous source systems. Surrogate keys ensure independence from such variability, preserve integration consistency across data domains, and improve join performance in analytical workloads due to their compact and indexed structure. This standardization is particularly important when dimensions are shared across multiple fact tables or when historical tracking strategies are applied.

MDM tables are managed through the same `MERGE` logic and temporal metadata conventions established in the ODS, ensuring consistent lineage and auditability across enrichment cycles.

OUT Layer (Output Staging Layer)

The OUT layer is the final staging step within L1. Its purpose is to translate enriched MDM data into structures fully aligned with the target analytical schema in L2, implementing the business-specific transformation logic that derives directly from functional requirements. These transformations may include multi-table joins to construct denormalized entities, calculated metrics and derived attributes, conditional classifications and mapping logic, and temporal adjustments aligned with reporting calendars.

The OUT table mirrors the exact structure of its L2 destination, ensuring seamless downstream loading. It is refreshed via a `TRUNCATE-INSERT` strategy to always contain the most recent and fully prepared dataset. By isolating business logic within this layer, the framework preserves architectural clarity: technical validation and integration are handled upstream in the OK and ODS stages, while semantic modeling for analytics is finalized immediately prior to publication.

At the conclusion of L1, all incoming incremental data has been validated, consolidated, enriched, and aligned with business requirements. Only records meeting all these criteria advance to Layer L2.

3.3.3 Layer L2 – Semantic Layer and Analytical Structuring

Layer L2 is the semantic and publication layer of the data platform. It receives the validated, consolidated, and enriched datasets produced in L1 and organizes them into analytics-ready dimensional structures designed for business consumption.

Data is modeled following a star schema approach [5], as discussed in Section 2.1.4: central fact tables capture measurable business events at a defined level of granularity, while surrounding dimension tables provide descriptive, categorical,

and hierarchical context. This structure reduces join complexity, enhances query efficiency, and reflects the natural way business users formulate analytical questions.

Layer L2 directly supports the analytical ecosystem of the organization. Business Intelligence platforms connect to the published fact and dimension tables to generate interactive reports and dashboards. Analysts perform ad hoc SQL queries on optimized dimensional schemas, while machine learning workflows extract structured feature datasets that benefit from the consistent governance and historization implemented upstream. Self-service analytics users interact with an intuitive, semantically coherent schema without requiring knowledge of the underlying ingestion and transformation processes.

It should be noted that Layer L2 falls outside the implementation scope of this thesis, which focuses on the L0 and L1 stages of the pipeline. Nevertheless, the architectural choices made at L0 and L1 — including surrogate key governance, MDM enrichment, and the OUT staging structure — are deliberately designed to anticipate and facilitate its future implementation. The proposed design is therefore conceived as a complete end-to-end architecture, even where its final stage has not yet been realized in the current Databricks deployment.

By isolating analytical modeling within a dedicated semantic layer, the framework shields end users from the technical complexity of ingestion and integration, while ensuring performance, governance, and semantic clarity throughout the analytical lifecycle.

3.4 Metadata Governance Layer

The framework relies on a comprehensive metadata governance layer to ensure traceability, execution control, and dynamic configurability across all stages of the pipeline [22]. This layer is implemented through three complementary tables — `FLOW_MANAGER`, `TABLE_MANAGER`, and `METADATA_MANAGER` — each serving a distinct governance purpose: process-level orchestration, table-level monitoring, and dynamic configuration management, respectively. Together they transform the pipeline into a controlled, self-regulating system in which governance, traceability, and reproducibility are enforced by design rather than by convention [22, 27].

3.4.1 FLOW_MANAGER

`FLOW_MANAGER` provides process-level orchestration, tracking the execution state of data flows across business domains and architectural layers. Table 3.1 illustrates its field structure.

Records are managed via a `MERGE` strategy keyed on `IDENTITY`, `GRP_NAME`, `NUM_LEVEL`, and `TRG_JOBID`. When a matching record is found, the fields `LOAD`,

Table 3.1: FLOW_MANAGER — field structure.

Field	Type	Description
`IDENTITY`	VARCHAR	Name of the execution environment in which the process is running (e.g. development, staging, production)
`NUM_LEVEL`	INTEGER	Logical processing layer: 0 = L0, 1 = L1, 2 = L2
`GRP_NAME`	VARCHAR	Functional domain or business area associated with the data flow
`TRG_JOBID`	VARCHAR	Unique identifier assigned to the current ETL execution, used to track its progress and lineage
`SRC_JOBID`	VARCHAR	Reference identifier for the upstream execution: holds the default value `19000101000000` at L0; assumes the value of `TRG_JOBID` at L1
`STATUS`	INTEGER	Process state: 0 = completed successfully, 1 = execution in progress, 3 = terminated with errors
`LOAD`	INTEGER	Data readiness flag for downstream consumption: 0 = already consumed, 1 = ready to be consumed, 2 = not yet ready, 5 = final level reached, no further consumption possible
`START_DATE`	TIMESTAMP	Execution start timestamp
`END_DATE`	TIMESTAMP	Execution end timestamp

STATUS, and END_DATE are updated in place, maintaining an accurate and continuously updated view of process execution states across the entire pipeline.

3.4.2 TABLE_MANAGER

TABLE_MANAGER provides fine-grained monitoring at the individual table level, capturing ingestion statistics for each processed dataset within every execution cycle. Table 3.2 illustrates its field structure.

Table 3.2: TABLE_MANAGER — field structure.

Field	Type	Description
IDENTITY	VARCHAR	Execution environment (development, staging, production)
LEVEL	INTEGER	Logical processing layer (0, 1, or 2)
GRP_NAME	VARCHAR	Functional or business domain
TABLE_NAME	VARCHAR	Identifier of the processed dataset
JOBID	VARCHAR	Reference to the ingestion execution cycle
NUM_ROWS	INTEGER	Number of records processed or loaded in the cycle
LOAD_DATE	TIMESTAMP	Timestamp of load completion

Unlike FLOW_MANAGER, TABLE_MANAGER follows an APPEND strategy, inserting a new record for every table processed in each execution cycle. This design preserves the complete execution history across time, supporting detailed performance analysis, volume trend monitoring, and operational auditing.

3.4.3 METADATA_MANAGER

METADATA_MANAGER provides a flexible key-value structure for dynamic pipeline configuration. Table 3.3 illustrates its field structure.

The key-value design is intentionally flexible: new parameters can be introduced at any time without schema modifications, accommodating client-specific requirements or evolving operational policies. Typical parameters include retention policies for rejected records in E$ tables, delta snapshot archival rules for DLT tables, notification lists for execution failures, parallelism limits for L0 extraction jobs, and thresholds for validation checks such as null ratio limits or anomaly detection criteria.

By externalizing all configuration from transformation code, the metadata governance layer enhances maintainability, reduces deployment complexity, and enables dynamic adaptation of pipeline behavior without modifying ETL scripts.

Table 3.3: METADATA_MANAGER — field structure.

Field	Type	Description
`IDENTITY`	VARCHAR	Execution environment (development, staging, production)
`GRP_NAME`	VARCHAR	Functional or business domain
`PARAM_NAME`	VARCHAR	Configuration parameter identifier
`PARAM_VALUE`	VARCHAR	Parameter value, interpreted according to the parameter type

3.5 Execution Model and Scheduling Rules

The layered architecture requires a controlled execution model to ensure consistency, reproducibility, and operational robustness across the pipeline. Because L0, L1, and L2 fulfill distinct and sequentially dependent roles, their orchestration must respect logical dependencies while optimizing computational efficiency. Execution is governed by metadata-driven scheduling rules that regulate concurrency, dependency management, and process monitoring at each stage.

At **Layer L0**, extraction jobs operate in parallel. Since this stage is limited to raw data acquisition and metadata initialization, and involves no semantic transformations, multiple source systems can be processed concurrently without risk to data integrity. Controlled parallelism significantly improves ingestion throughput in environments with high data volumes and heterogeneous source systems.

At **Layer L1**, execution follows a dependency-aware serialization model. Because this layer applies validation controls, integration rules, historization mechanisms, and business transformations, unconstrained concurrency could introduce referential inconsistencies or nondeterministic outcomes. Transformations are therefore executed in a serialized or dependency-constrained sequence, ensuring that validation checkpoints, error recycling, and ODS updates proceed in a coherent and fully auditable order.

At **Layer L2**, execution is strictly conditional upon the successful completion of all relevant L1 processes. Only fully validated and integrated OUT datasets are eligible for publication into dimensional structures. Within L2, selective parallelism may be applied where structural independence permits — for instance, when dimension tables can be populated independently without violating foreign key constraints or SCD logic [5].

This hierarchical dependency model preserves the logical autonomy of each processing stage while enabling efficient resource utilization. The orchestration is implemented entirely through the metadata tables described in Section 3.4, which track execution status, dependency chains, and operational parameters. As a result, the ETL pipeline functions as a self-regulating and fully auditable system, capable of managing complex interdependencies while guaranteeing deterministic and reproducible execution cycles.

Chapter 4

Framework Implementation on Databricks

This chapter describes the practical implementation of the ETL framework on Databricks, covering the full data pipeline from raw ingestion through to the Operational Data Store (ODS) tables of Layer L1. The implementation was intentionally scoped to these layers in order to prioritize a thorough study of the Databricks platform — its architectural components, governance mechanisms, and data engineering capabilities — and to evaluate its feasibility within a structured enterprise framework. Layer L2 and the MDM stage remain structurally compatible with the existing architecture and are identified as natural extensions for future development, as the architectural alignment between the OUT layer structures produced in L1 and the target L2 schemas has been verified at the design level.

The chapter is organized as follows. Section 4.1 describes the data ingestion environment, including Unity Catalog configuration and the logical organization of data assets. Section 4.2 describes the data preparation and quality enforcement procedures applied during the ingestion phase, including raw standardization and type enforcement. Section 4.3 covers the full Layer L0 and Layer L1 pipeline implementations, including raw ingestion, data standardization, change data capture, business rule enforcement, error recycling, and ODS consolidation. Section 4.4 details the audit and governance infrastructure. Section 4.5 describes the Databricks Jobs configuration and the end-to-end workflow execution model.

4.1 Data Ingestion and Storage

4.1.1 Catalog and Schema Configuration

Before ingesting any data, the Databricks environment was structurally configured to reflect the logical architecture of the ETL framework. Data assets were organized using Unity Catalog [30], which provides centralized governance and hierarchical organization of database objects.

In Databricks, a *catalog* represents the highest-level logical container in the data management hierarchy. It serves as a governed namespace that groups schemas and their associated objects while enforcing access control policies and ensuring consistency across environments. A dedicated catalog was created to host the entire ETL framework, thereby isolating the project within a controlled and traceable domain.

Within the catalog, multiple schemas were defined to mirror the three-layer architecture of the framework:

- Layer L0, dedicated to raw data ingestion and initial staging structures;
- Layer L1, containing validated, transformed, and historized datasets;
- A schema reserved for Layer L2 analytical structures, provisioned at the design level to facilitate future extension although not yet populated within the scope of this thesis;
- An additional schema supporting advanced analytics and machine learning artifacts.

A *schema* in Databricks is a logical subdivision of a catalog that groups related database objects such as tables and views. The hierarchical organization (catalog $\rightarrow$ schema $\rightarrow$ table) ensures modularity, clarity, and governance across different processing stages [30].

In addition to schemas, managed volumes [30] were created within the ingestion layer. A *volume* is a secure storage location governed by Unity Catalog that allows structured management of files and non-tabular data. Volumes enable direct interaction with raw files while preserving centralized access control and auditing mechanisms.

The initialization of the environment was performed programmatically using SQL statements that create or reset catalogs, schemas, volumes, and metadata structures. This controlled initialization ensures a deterministic execution context, allowing the entire pipeline to be re-executed from scratch for testing, validation, or reproducibility purposes. An excerpt of the initialization statements is reported below:

```sql
-- Creation of the main catalog
CREATE CATALOG IF NOT EXISTS project_catalog;

-- Creation of Layer L0 (Raw Ingestion)
CREATE SCHEMA IF NOT EXISTS project_catalog.l0_layer;

-- Creation of Layer L1 (Integration and Validation)
CREATE SCHEMA IF NOT EXISTS project_catalog.l1_layer;

-- Creation of a governed storage volume for the landing zone
CREATE VOLUME IF NOT EXISTS project_catalog.l0_layer.landing_zone;
```

Listing 4.1: Initialization of Unity Catalog structure

The use of conditional clauses such as `IF NOT EXISTS` guarantees idempotent execution. In other words, the statements can be executed multiple times without generating duplicate objects or unintended side effects. This property enables safe reinitialization of the environment while preventing accidental object duplication or structural inconsistencies.

4.1.2 Raw Data Ingestion and Landing Zone Setup

The datasets used in this study were provided in CSV format, including a header row and using the semicolon (;) as field separator. These files represent structured tabular data extracted from operational systems and constitute the initial input of the ETL pipeline. A dedicated landing zone volume was established within Layer L0 to host incoming raw files. For the purpose of this implementation, the CSV datasets were manually uploaded into the landing zone through the Databricks interface. This approach was intentionally adopted to maintain full control over the experimental setup and to ensure reproducibility during development and testing phases.

Although manual ingestion was suitable for prototyping and controlled experimentation, production-grade environments typically rely on automated ingestion mechanisms. In enterprise contexts, data sources are often integrated through cloud object storage services, streaming platforms, or enterprise data integration tools. Databricks supports these scenarios through structured ingestion capabilities, including Auto Loader [10] and native connectivity to cloud storage systems, enabling scalable and incremental data acquisition.

Overall, the ingestion layer establishes a governed and modular storage foundation, aligning the physical organization of raw data with the logical architecture

of the ETL framework and preparing datasets for subsequent transformation, validation, and integration processes.

4.1.3 Incremental Loading with Auto Loader

To enable scalable and incremental ingestion, the pipeline leverages Databricks Auto Loader, a cloud-native file ingestion mechanism built on Structured Streaming. Auto Loader continuously monitors a designated storage location and incrementally processes only newly arrived or previously unprocessed files, ensuring efficient and fault-tolerant data acquisition.

In this implementation, Auto Loader is configured to monitor the landing zone volume within Layer L0. When the pipeline is triggered, it automatically detects CSV files that have not yet been processed and ingests them in streaming mode. This mechanism eliminates the need for manual file tracking and guarantees exactly-once processing semantics through checkpointing and metadata management.

The ingestion logic is implemented using Delta Live Tables (DLT) [28], which allows declarative definition of streaming tables. The following excerpt illustrates the core configuration used to read CSV files through Auto Loader:

```
spark.readStream \
    .format("cloudFiles") \
    .option("cloudFiles.format", "csv") \
    .option("cloudFiles.schemaLocation", schema_path) \
    .option("cloudFiles.inferColumnTypes", "false") \
    .option("header", "true") \
    .option("sep", ";") \
    .option("cloudFiles.schemaEvolutionMode", "rescue") \
    .load(data_path)
```

Listing 4.2: Auto Loader configuration for CSV ingestion

The `cloudFiles` format activates Auto Loader, enabling incremental file discovery. The `schemaLocation` option specifies a managed path where schema metadata is stored, allowing the system to track structural evolution across different file batches. Since source files are provided in CSV format with header rows and semicolon delimiters, the corresponding parsing options are explicitly defined.

Particular attention is given to schema evolution. By setting the `cloudFiles.schemaEvolutionMode` parameter to *rescue*, unexpected or newly introduced columns are automatically captured in a dedicated field instead of causing pipeline failures. This approach enhances robustness and operational resilience, especially in scenarios where upstream systems may introduce structural changes without prior notification.

Beyond file ingestion, additional transformations are applied to standardize column naming conventions. Column names are normalized to lowercase, special

characters are removed, and duplicates are automatically disambiguated through suffixes. This normalization ensures compatibility with downstream processing layers and prevents ambiguities during subsequent transformations. Furthermore, ingestion metadata is enriched by attaching the source file path and execution identifiers to each record, in support of full traceability, auditability, and lineage tracking across the ETL lifecycle.

The adoption of Auto Loader transforms the ingestion layer from a static batch-oriented mechanism into a continuously incremental processing system. Even though the experimental setup relies on manually uploaded files, the architecture is inherently designed for production environments where new data may arrive asynchronously and at scale. In summary, incremental loading with Auto Loader ensures scalability, resilience to schema drift, and operational efficiency, reinforcing the metadata-driven and modular nature of the implemented ETL framework.

4.2 Raw Standardization and Type Enforcement

Although the following processes are technically part of Layer L0, they are presented in a dedicated section to highlight their methodological relevance within the overall framework. During the initial ingestion phase, several data quality issues were identified in the source CSV files. These anomalies required the introduction of an intermediate standardization step before progressing to higher integration layers.

In the DSO framework specification described in Chapter 3, a single staging table per dataset was originally designed to host ingested data. However, empirical analysis of the provided datasets revealed inconsistencies introduced during CSV export procedures. As a result, the staging layer was redesigned to include two distinct tables: *STG_RAW* and *STG_CLEAN*. This architectural refinement enhances robustness, traceability, and data governance.

4.2.1 Raw Data Preservation

The *STG_RAW* tables store data exactly as extracted from the CSV files, without semantic interpretation or strict type enforcement. All columns are initially treated as strings. This design choice prevents unintended type coercion during ingestion and preserves the original content of the source files.

During preliminary inspection of the datasets, several structural inconsistencies were identified:

- Identifier fields containing artificial decimal suffixes (e.g., `12345.0`), which would otherwise be interpreted as floating-point values;
- Date fields exported with trailing decimal markers;

- Numeric fields expressed using locale-specific formats (e.g., `100.000,00`);
- Inconsistent capitalization and irregular whitespace.

If automatically inferred, these patterns could lead to incorrect type assignments, parsing errors, or silent data corruption. To mitigate this risk, the ingestion logic explicitly disables automatic type inference and enforces string-based ingestion.

By preserving all values as raw strings, the *STG_RAW* layer functions as a tabular and queryable representation of the original files. This ensures full data lineage and allows the original content to remain accessible even if downstream cleansing logic requires revision. The approach aligns with lakehouse principles that advocate for immutable raw data zones [6].

4.2.2 Data Cleansing and Type Enforcement

Following raw ingestion, a second transformation step generates the *STG_CLEAN* tables. Although still part of Layer L0, this stage introduces controlled semantic interpretation and deterministic data type enforcement.

The cleansing logic is implemented within the same Delta Live Tables (DLT) pipeline and operates in streaming mode. The transformation is dynamically generated through a factory function that creates one clean table per dataset based on a predefined column mapping configuration. This mapping specifies the intended semantic type of each column (e.g., *bigint*, *double*, *int*, *date*) and was defined after systematic inspection of the source CSV files.

The transformation performs controlled casting, normalization of numeric formats, case standardization, and multi-pattern date parsing. A simplified excerpt of the implementation is reported below:

```
for col_name in df_raw.columns:

    col_expr = F.col(col_name)
    c_upper = col_name.upper()

    # BIGINT normalization
    if c_upper in set_bigint:
        clean_val = F.regexp_replace(F.trim(col_expr), r"\.0$", "")
        col_expr = clean_val.cast("bigint")

    # DOUBLE normalization (European vs standard format)
    elif c_upper in set_double:
        str_col = F.regexp_replace(F.trim(col_expr), r"^\+", "")
        col_expr = F.when(
```

```
15              str_col.contains(","),
16              F.regexp_replace(
17                  F.regexp_replace(str_col, r"\.", ""),
18                  ",", "."
19              ).cast("double")
20          ).otherwise(str_col.cast("double"))
21
22      # DATE parsing with fallback
23      elif c_upper in set_date:
24          clean_str = F.regexp_replace(F.trim(col_expr), r"\.0
    $", "")
25          col_expr = F.coalesce(
26              F.to_date(clean_str, "yyyyMMdd"),
27              F.to_date(clean_str, "dd/MM/yyyy"),
28              F.to_date(clean_str, "yyyy-MM-dd")
29          )
30      final_expressions.append(col_expr.alias(col_name))
```

Listing 4.3: Core type enforcement logic in STG_CLEAN

The cleansing phase performs the following controlled operations:

- Removal of artificial decimal suffixes in identifier fields;
- Normalization of heterogeneous numeric formats (European and standard notation);
- Explicit and deterministic casting of numeric attributes;
- Multi-pattern date parsing with fallback logic;
- Standardization and trimming of string values.

This approach differs from automatic schema inference by enforcing explicit semantic interpretation rules. All transformations are deterministic, reproducible, and auditable within the DLT pipeline definition.

The coexistence of *STG_RAW* and *STG_CLEAN* tables reflects a governance-oriented architectural design. The raw layer guarantees immutability and full traceability of source data, while the clean layer ensures structural correctness and analytical consistency. If cleansing rules require modification, transformations can be re-executed without re-ingesting source files, preserving reproducibility and minimizing operational risk.

4.3 ETL Pipeline Implementation

4.3.1 Delta Live Tables Overview

The ETL pipeline was implemented using Delta Live Tables (DLT), a declarative data engineering framework provided by Databricks for building reliable, maintainable, and scalable data pipelines. DLT enables the definition of data transformations through Python or SQL while automatically managing orchestration, dependency resolution, execution planning, and monitoring.

In contrast to traditional imperative ETL scripts, DLT adopts a declarative paradigm: developers define target tables and their transformation logic, and the platform automatically determines execution order based on lineage dependencies. When the pipeline is executed, Databricks generates a directed acyclic graph (DAG) that visually represents table dependencies, data flow, and processing stages. This graphical representation enhances transparency, traceability, and debugging capabilities.

DLT supports three primary dataset types, all of which were leveraged in this implementation to model different processing semantics:

- **Streaming Tables**: continuously updated tables built on Structured Streaming. They ingest and append new records incrementally while preserving historical data across pipeline executions, making them suitable for ingestion and change data capture scenarios.
- **Materialized Views**: batch-oriented tables recomputed at each pipeline execution. Unlike streaming tables, they typically overwrite previous content and are designed for deterministic transformations and consolidated outputs.
- **Views**: logical datasets that do not physically materialize data but instead provide virtual transformations over upstream tables. Views are primarily used to modularize complex transformations, enhance readability, and separate intermediate business logic from persistent storage layers.

Streaming tables were adopted when historical tracking was required, particularly in the staging layers and change data capture logic, where incremental ingestion and persistence of historical records were essential. Materialized views were used when the structural definition of a dataset needed to remain stable while its content was deterministically recomputed at each pipeline execution. Views were employed to encapsulate intermediate transformation logic without persisting additional physical tables.

All pipelines were implemented entirely in Python using the `import dlt` module. Although DLT also supports SQL-based definitions, Python was selected to enable modular programming, reusable functions, and metadata-driven configuration. To

improve maintainability and logical separation of concerns, the overall ETL flow was divided into two distinct pipelines, as described in detail in the subsequent sections of this chapter.

4.3.2 Implementation of Layer L0 Pipelines

Layer L0 was implemented as a modular Python-based Delta Live Tables pipeline structured across five coordinated modules: `MetaData.py`, responsible for managing run-level metadata; `STG_raw.py`, handling raw streaming ingestion; `STG_clean.py`, implementing controlled type enforcement and cleansing logic; `DLTstep.py`, orchestrating the variation logic; and `mainL0.py`, acting as the pipeline entry point and configuration layer. Each module encapsulates a specific responsibility, enabling reusability and clear separation between ingestion, cleansing, metadata management, and change detection.

Metadata Management The pipeline execution is driven by a run identifier (*jobid*) that uniquely identifies each execution. To manage this information, two DLT tables are created:

- *run_metadata*: a batch snapshot containing the current job identifier;
- *run_metadata_his*: a streaming table maintaining the full historical log of executions.

A simplified excerpt of the metadata logic is shown below:

```
1  @dlt.table(name="run_metadata")
2  def run_metadata():
3      return spark.read.table("audit_schema.
       run_metadata_signal")
4
5
6  @dlt.table(name="run_metadata_his")
7  def run_metadata_his():
8      return (
9          spark.readStream
10             .option("ignoreChanges", "true")
11             .table("audit_schema.run_metadata_signal")
12             .withColumn("ingestion_ts", current_timestamp())
13     )
```

Listing 4.4: Run metadata materialization in DLT

The batch table provides contextual information for the current execution, while the streaming table preserves historical traceability across runs. This separation ensures both operational clarity and full auditability.

Staging Tables As described in Section 4.2, the staging layer consists of two streaming tables per dataset:

- *STG_RAW*: stores raw ingested data exactly as read from CSV files;
- *STG_CLEAN*: applies cleansing and controlled type enforcement.

Both tables are defined as streaming tables in order to preserve historical consistency and incremental processing semantics. This design ensures that each pipeline execution contributes new data without overwriting previously processed records.

Change Data Capture To detect variations between consecutive pipeline executions, a custom Change Data Capture (CDC) mechanism was implemented at the boundary between the cleansing stage and the integration layer. Rather than relying on built-in Delta comparison utilities, the system adopts a reproducible, execution-driven comparison strategy that operates exclusively on pipeline-level identifiers.

Each pipeline run is uniquely identified by a *jobid*, which acts as a discrete temporal reference. The CDC logic is encapsulated in the function `compute_cdc_by_jobid`, which accepts the full historical dataset, the business key, and two metadata DataFrames: `curr_meta_df`, containing the current run's metadata, and `his_meta_df`, containing the accumulated history of all past executions.

The current execution is resolved by extracting the *jobid* from `curr_meta_df`, while the previous execution is identified dynamically by selecting the maximum *jobid* strictly lower than the current one from the historical metadata. This ensures that comparisons always reflect strictly adjacent pipeline states, independent of ingestion timestamps or system-generated metadata.

```
1  df_curr_id = curr_meta_df.select(F.col("jobid").alias("
       curr_id"))
2
3  prev_jobid_val = his_meta_df.alias("h").join(
4      df_curr_id.alias("c"),
5      F.col("h.jobid") < F.col("c.curr_id")
6  ).select(F.max(F.col("h.jobid")).alias("prev_id"))
7
8  df_curr = df.join(df_curr_id, df.jobid == df_curr_id.curr_id
9            ).drop("curr_id")
10 df_prev = df.join(prev_jobid_val, df.jobid == prev_jobid_val
       .prev_id
11           ).drop("prev_id")
```

Listing 4.5: Identification of current and previous execution states

After filtering, the auxiliary join columns (`curr_id`, `prev_id`) are dropped to keep the schema clean for downstream operations.

Column Selection and Technical Attribute Exclusion

Before performing the comparison, the set of attributes subject to change detection is determined dynamically. A predefined list of technical columns — `jobid`, `ins_time`, `source_file`, `ingestion_ts`, and `file_path` — is explicitly excluded alongside the business key columns. The remaining columns form the `compare_cols` list, which represents the semantically meaningful attributes subject to change detection.

```
tech_cols = ["jobid", "ins_time", "source_file",
             "ingestion_ts", "file_path"]
exclude_cols = {c.lower() for c in business_key + tech_cols}
compare_cols = [c for c in df.columns if c.lower() not in
   exclude_cols]
```

Listing 4.6: Dynamic construction of the comparison column set

This design prevents false positives that would otherwise arise from purely technical changes unrelated to business semantics.

Symmetric Set-Based Comparison Logic

The detection logic is grounded in relational set algebra and is implemented through a combination of `left_anti` and `inner` joins. Three categories of change are identified and handled as shown in Listing 4.7.

```
# 1. New records (present in T but not in T-1)
df_new = df_curr.alias("c").join(
    df_prev.alias("p"), on=business_key, how="left_anti"
).withColumn("flag_neg", F.lit(0))

# 2. Deleted records (present in T-1 but not in T)
df_deleted = df_prev.alias("p").join(
    df_curr.alias("c"), on=business_key, how="left_anti"
).crossJoin(df_curr_id) \
 .withColumn("flag_neg", F.lit(1)) \
 .withColumn("jobid", F.col("curr_id")).drop("curr_id")

# 3. Modified records: composite difference condition
diff_cond = F.lit(False)
for c in compare_cols:
    diff_cond |= (
        F.coalesce(F.col(f"c.{c}").cast("string"), F.lit("\
   u2205")) !=
        F.coalesce(F.col(f"p.{c}").cast("string"), F.lit("\
   u2205"))
```

```
    )

df_changed_base = df_curr.alias("c").join(
    df_prev.alias("p"), on=business_key, how="inner"
).filter(diff_cond)

# Reversal of the previous state (logical closure)
df_changed_old = df_changed_base.select("p.*") \
    .crossJoin(df_curr_id) \
    .withColumn("flag_neg", F.lit(1)) \
    .withColumn("jobid", F.col("curr_id")).drop("curr_id")

# Insertion of the updated current state
df_changed_new = df_changed_base.select("c.*") \
    .withColumn("flag_neg", F.lit(0))
```

Listing 4.7: CDC classification logic and record bifurcation

Category 1 — Insertions

Records present in the current execution but absent in the previous one are identified via a `left_anti` join from T against $T-1$. This operation implements set subtraction declaratively, without requiring explicit null checks. Entities classified as newly inserted are assigned $\mathit{flag\_neg} = 0$, indicating an active, open record version.

Category 2 — Deletions

The symmetric `left_anti` join in the opposite direction identifies records that existed in $T-1$ but are absent in T. A compensating record is generated with $\mathit{flag\_neg} = 1$, representing a logical closure. The *jobid* is reassigned to the current execution identifier via a `crossJoin` with the single-row metadata dataset `df_curr_id`. This reassignment ensures that downstream merge operations interpret the deletion as temporally posterior to the original insertion, preserving deterministic conflict resolution. The `crossJoin` is safe because the metadata dataset is guaranteed to contain exactly one row per execution, as enforced by the overwrite-mode write described in Section 4.4.1.

Category 3 — Modifications

Records present in both T and $T-1$ are joined via an `inner` join on the business key. A composite difference condition (`diff_cond`) is constructed programmatically by iterating over `compare_cols` and accumulating logical `OR` predicates. Each attribute is explicitly cast to `string` and normalized using the `coalesce()` function, replacing `NULL` values with the symbol ∅ (`U+2205`) used as a sentinel placeholder before comparison. This specific character was chosen because it is semantically unambiguous and extremely unlikely to appear as a legitimate business value, thus

eliminating the risk of false equality between a null field and an empty string. This normalization step resolves the SQL three-valued logic issue [36], according to which the expression `NULL <> NULL` evaluates to *unknown* rather than *false*. Without this precaution, certain attribute changes involving `NULL` values could remain undetected.

When a difference is detected, the engine generates two compensating records: a reversal of the previous state ($flag\_neg = 1$) and an insertion of the updated current state ($flag\_neg = 0$). This symmetric Delete+Insert pattern mirrors a Slowly Changing Dimension Type 2 (SCD2) approach [5], guaranteeing full historical traceability without in-place updates.

Output Consolidation

The four partial DataFrames — new records, deleted records, updated old states, and updated new states — are consolidated into a single output via `unionByName` with `allowMissingColumns=True`, which tolerates potential schema differences across datasets. Finally, the `ins_time` field is overwritten with the current timestamp to reflect the actual CDC processing time rather than the original ingestion time.

Dual-Table CDC Output Architecture

The CDC tables are generated through a parametric factory function `create_cdc_tables`, which accepts a dataset prefix and the corresponding business key and registers two Delta Live Tables via the `@dlt.table` decorator.

```
1  def create_cdc_tables(prefix: str, business_key: list):
2      p_lower = prefix.lower()
3
4      @dlt.table(name=f"dlt_{p_lower}")
5      def cdc_view():
6          return compute_cdc_by_jobid(
7              dlt.read(f"stg_{p_lower}_clean"),
8              business_key,
9              dlt.read("run_metadata"),
10             dlt.read("run_metadata_his")
11         )
12
13     @dlt.table(name=f"dlt_his_{p_lower}",
14                comment=f"Full CDC history for {prefix}")
15     def cdc_history():
16         return (
17             spark.readStream
18                 .option("ignoreChanges", "true")
19                 .option("allowMissingColumns", "true")
20                 .table(f"LIVE.dlt_{p_lower}")
```

```
21                 )
```

Listing 4.8: Parametric DLT table factory for CDC output

For each dataset, the factory produces two tables. The first, *dlt_<dataset>*, is a materialized batch table fully recomputed at each execution and containing only the delta detected for the current run. The second, *dlt_his_<dataset>*, is a streaming table that incrementally accumulates all historical deltas by reading from the batch table via `readStream`. The `ignoreChanges = true` option is essential: since the source table is overwritten at each execution, this flag instructs the streaming reader to treat recomputed data as valid new input rather than raising an error, enabling correct incremental accumulation over time. The `allowMissingColumns = true` option further ensures robustness against schema evolution across pipeline versions.

Semantic Interpretation of *flag_neg*

The *flag_neg* attribute encodes logical state transitions rather than physical deletions: a value of 0 denotes an active, open record version, while a value of 1 denotes a logically closed or reversed version. This encoding enables complete reconstruction of entity history at any point in time and remains fully compatible with downstream merge-based integration mechanisms.

Design Rationale

The symmetric execution-driven CDC model was adopted for the following reasons. Change detection is governed entirely by pipeline execution identifiers, making the process deterministic and independent of built-in Delta change tracking. Explicit *jobid*-based sequencing ensures consistent yesterday–today and today–yesterday semantics across all runs. The compensating record strategy produces an immutable, append-only audit trail of all state transitions, enhancing governance and auditability. Finally, exposing changes as explicit insertions and reversals simplifies the downstream merge logic in the Operational Data Store by avoiding in-place updates entirely.

Pipeline Entry Point The orchestration of the Layer L0 pipeline is governed by a metadata-driven approach implemented in the `mainL0.py` module. This file constitutes the sole entry point of the Delta Live Tables pipeline and is specified as the source script in the pipeline configuration. Upon execution, Databricks parses this module, resolves inter-table dependencies through lineage analysis, and automatically constructs a directed acyclic graph (DAG) representing the complete execution workflow. The resulting DAG for the Layer L0 pipeline is illustrated in Figure 4.1.

Rather than hardcoding transformation logic for each dataset, the pipeline relies on a declarative configuration structure in which each source is defined through a

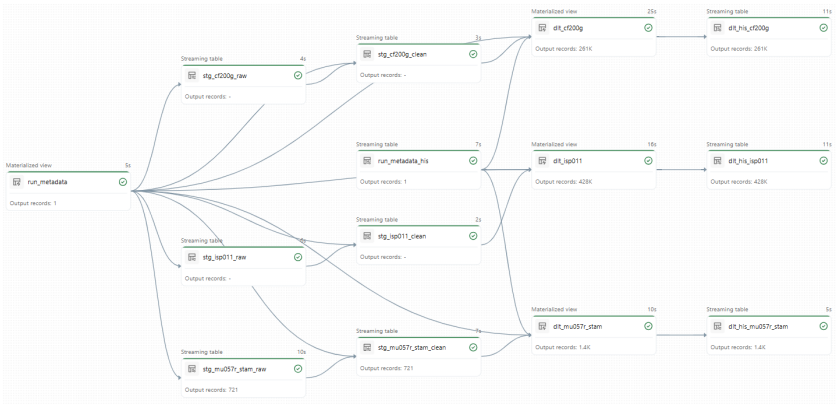

Figure 4.1: DAG representation of the Layer L0 pipeline.

parameter dictionary covering raw ingestion, cleansing, and CDC computation.

Dataset Configuration Dictionary

Each dataset is represented as a Python dictionary within the `datasets` list. The structure is uniform across all sources, regardless of their schema complexity. An illustrative example of a single dataset entry is shown below:

```
{
    "prefix": "<DATASET_PREFIX>",
    "folder": "<SOURCE_FOLDER>",
    "format": "csv",
    "sep": ";",
    "business_key": ["<key_column_1>", "<key_column_2>"],
    "cleaning": {
        "bigint": ["<col_a>", "<col_b>"],
        "double": ["<col_c>", "<col_d>"],
        "int":    ["<col_e>"],
        "date":   ["<col_f>", "<col_g>"]
    }
}
```

Listing 4.9: Example dataset configuration entry

Automated Pipeline Orchestration

Once the configuration is defined, the module initializes the metadata tables and iterates over the `datasets` list, invoking three factory functions in sequence for each dataset:

```
from DLTstep import create_cdc_tables
from MetaData import create_metadata_tables
from STG_raw import create_stg_raw_table
from STG_clean import create_stg_clean_table

create_metadata_tables()

for ds in datasets:
    create_stg_raw_table(
        prefix      = ds["prefix"],
        data_path   = f"{BASE_DATA_PATH}/{ds['folder']}",
        schema_path = f"{BASE_SCHEMA_PATH}/{ds['prefix']}",
        file_format = ds["format"],
        sep         = ds.get("sep", ";")
    )
    create_stg_clean_table(
        prefix         = ds["prefix"],
        column_mapping = ds["cleaning"]
```

```
    )
    create_cdc_tables(
        prefix       = ds["prefix"],
        business_key = ds["business_key"]
    )
```

Listing 4.10: Metadata initialization and dataset orchestration loop

The `create_metadata_tables()` call is executed once before the loop, registering the `run_metadata` and `run_metadata_his` tables that track execution context across pipeline runs. No explicit execution ordering needs to be specified: Delta Live Tables automatically resolves inter-table dependencies by analyzing the `dlt.read()` and `spark.readStream` references declared within each factory function.

4.3.3 Implementation of Layer L1 Pipelines

Following the successful execution of the L0 ingestion layer, the ETL workflow proceeds to Layer L1, which is responsible for semantic validation, business rule enforcement, and consolidation into an Operational Data Store (ODS). While Layer L0 guarantees structural normalization, incremental ingestion, and historical traceability, Layer L1 introduces domain-aware validation and controlled integration. In architectural terms, L1 represents the transition from technical data governance to business-oriented data reliability.

The L1 pipeline is structured into four modular components — `metadata.py`, `business_rules.py`, `ok_err.py`, and `ods.py` — orchestrated through a centralized execution script (`main_l1.py`). This modular decomposition promotes separation of concerns, maintainability, and reusability across heterogeneous datasets.

Metadata Synchronization Between Layers To ensure continuity between L0 and L1, a metadata synchronization mechanism is implemented in `metadata.py`. Since L0 operates using a job-based execution model governed by a *jobid*, L1 must inherit the same execution identifier to maintain lineage consistency and referential coherence across layers.

Rather than registering a materialized DLT table — which would conflict with the ownership already established by L0 — the L1 metadata module declares a `dlt.view`. This design choice is deliberate: a view provides each pipeline with its own local reference to the shared execution signal without creating ownership conflicts or duplicating physical storage. The underlying data is read directly from the audit infrastructure table, which is populated at each run.

```
@dlt.view(name="run_metadata")
def run_metadata():
```

```
    return spark.read.table(
        "<catalog>.<audit_schema>.run_metadata_signal"
    )
```

Listing 4.11: L1 metadata synchronization via DLT view

This view acts as a contextual anchor for all downstream L1 transformations. By referencing the same physical audit signal as L0, the pipeline guarantees that both layers operate under the same execution context, ensuring traceability and cross-layer reproducibility.

Business Rule Definition and Data Quality Enforcement Layer L1 introduces a declarative business validation framework implemented in `business_rules.py`. Each dataset is associated with a dedicated validation function that receives a DataFrame and augments it with boolean rule columns, each encoding a specific constraint. The validation framework covers four categories of rules: primary key integrity, temporal coherence across event sequences, numerical and financial consistency, and cross-field logical dependencies. The following example illustrates the general pattern:

```
def business_rules_dataset(df):

    # Primary key integrity: all key fields must be non-null
    df = df.withColumn(
        "rule_pk",
        F.col("key_col_1").isNotNull() &
        (F.trim(F.col("key_col_1")) != "") &
        F.col("key_col_2").isNotNull()
    )

    # Financial consistency: funded amount must be present
    df = df.withColumn(
        "rule_amounts",
        F.when(F.col("funded_amount").isNotNull(),
               F.lit(True)).otherwise(F.lit(False))
    )

    # Temporal coherence: maturity date must not precede
    # origination date
    df = df.withColumn(
        "rule_date_chain",
        F.when(
            F.col("maturity_date").isNotNull() &
            F.col("origination_date").isNotNull(),
```

```
            F.col("maturity_date") >= F.col("origination_date")
        ).otherwise(F.lit(True))
    )

    # Non-negativity: revenue figures must be non-negative
    df = df.withColumn(
        "rule_positive_amount",
        F.coalesce(F.col("revenue").cast("double") >= 0,
                   F.lit(True))
    )

    # Composite rule: all individual rules must pass
    return df.withColumn(
        "all_rules_passed",
        F.coalesce(F.col("rule_pk"), F.lit(False)) &
        F.coalesce(F.col("rule_amounts"), F.lit(False)) &
        F.coalesce(F.col("rule_date_chain"), F.lit(False)) &
        F.coalesce(F.col("rule_positive_amount"), F.lit(False))
    )
```

Listing 4.12: Illustrative business rule function structure

Each rule evaluates to a boolean column, and the composite `all_rules_passed` column is computed as the logical conjunction of all individual rules. The use of `coalesce` ensures that any rule returning `NULL` — due to upstream missing values or SQL three-valued logic — is conservatively treated as a failure rather than a pass, preventing invalid records from propagating undetected. It should be noted that in a production context, business rules of this nature would require formal validation with domain stakeholders and compliance teams to ensure regulatory and operational correctness.

Error Recycling and OK/ERR Separation A key architectural decision in L1 is the introduction of a dual-table validation mechanism implemented in `ok_err.py`. Valid records are written to `ok_<dataset>` tables, while invalid records are preserved in `err_storico_<dataset>` tables. Unlike simplistic rejection strategies that permanently discard failing records, this implementation supports *error recycling*: previously rejected records are reintroduced into subsequent pipeline runs and revalidated, allowing them to pass if upstream data corrections have occurred in the interim. This design prevents irreversible data loss and supports iterative data quality improvement over time.

The logic proceeds through three steps. First, the current CDC delta is read from the L0 layer and marked with an ingestion priority of 1. Second, any records

previously stored in the error table are retrieved and marked with a priority of 0. The two datasets are combined via `unionByName`, and a deduplication window function resolves conflicts by retaining the highest-priority version of each record identified by its business key — ensuring that freshly ingested data always supersedes stale error records.

```
df_new = spark.read.table(f"{source_layer}.dlt_{p_lower}") \
              .withColumn("_priority", F.lit(1))

if err_table_exists:
    df_old_err = spark.sql(
        f"SELECT *, 0 as _priority "
        f"FROM {target_layer}.err_storico_{p_lower}"
    )
    combined = df_new.unionByName(df_old_err,
                                  allowMissingColumns=True)
    w = Window.partitionBy(*business_key).orderBy(
        F.col("_priority").desc()
    )
    df_dedup = combined \
        .withColumn("_rn", F.row_number().over(w)) \
        .filter(F.col("_rn") == 1) \
        .drop("_rn", "_priority")
else:
    df_dedup = df_new.drop("_priority")
```

Listing 4.13: Error recycling and deduplication logic

After deduplication, business rules are applied to the consolidated dataset. The *jobid* and `ins_time` fields are then refreshed via a `crossJoin` with the current run metadata, ensuring that recycled records carry the execution identifier of the run in which they were last evaluated rather than that of their original ingestion. This guarantees temporal consistency throughout the layer.

```
return (
    business_rules_func(df_dedup)
    .crossJoin(current_job_meta)
    .withColumn("jobid",    F.col("current_run_jobid"))
    .withColumn("ins_time", F.current_timestamp())
    .drop("current_run_jobid")
    .withColumn("all_rules_passed",
                F.coalesce(F.col("all_rules_passed"),
                           F.lit(False)))
)
```

Listing 4.14: Metadata refresh and rule application

The validated dataset is then bifurcated into two tables. Records where `all_rules_passed = True` are written to `ok_<dataset>`, with all intermediate rule columns dropped to preserve a clean schema. Records where `all_rules_passed = False` are written to `err_storico_<dataset>` along with their diagnostic rule columns, enabling downstream inspection and root cause analysis.

Operational Data Store Consolidation The final stage of the L1 pipeline consolidates validated records into an Operational Data Store, implemented in `ods.py`. The ODS follows a Slowly Changing Dimension Type 1 (SCD1) strategy [5]: when a record matching an existing key is detected, its business attributes are overwritten with the most recent values; when no match is found, a new record is inserted. This design reflects the operational nature of the ODS layer, which prioritizes current-state representation. Full historical preservation is already guaranteed by the CDC mechanisms in L0 and is therefore not duplicated here.

The implementation is structured into four steps. The first step materializes a deduplicated preparation table from the validated `ok_<dataset>` table. Within this step, a window function resolves any residual duplicates by ordering records by descending *jobid* and ascending *flag_neg* — ensuring that the most recent active version of each entity is retained. Alongside standard business columns, two pairs of audit fields are managed: `JOBID` and `INS_TIME`, which record the execution context and timestamp of the first insertion and are never overwritten, and `JOBID_UPDATE` and `UPDATE_TIME`, which are refreshed at every merge to reflect the most recent modification.

```
w = Window.partitionBy(*pk_cols).orderBy(
    F.col("JOBID").desc(),
    F.coalesce(F.col("flag_neg"), F.lit(1)).asc()
)
df_dedup = df_with_meta \
    .withColumn("_rn", F.row_number().over(w)) \
    .filter(F.col("_rn") == 1).drop("_rn")

if ods_exists:
    df_existing = spark.read.table(f"LIVE.ods_{prefix}") \
        .select(*pk_cols,
                F.col("JOBID").alias("original_jobid"),
                F.col("INS_TIME").alias("original_ins_time")
   )
    df_prepared = (
        df_dedup.drop("JOBID", "INS_TIME")
        .join(df_existing, on=pk_cols, how="left")
        .withColumn("JOBID",
```

```
            F.coalesce(F.col("original_jobid"),
                       F.col("curr_jobid")))
        .withColumn("INS_TIME",
            F.coalesce(F.col("original_ins_time"),
                       F.col("curr_ts")))
        .withColumn("JOBID_UPDATE", F.col("curr_jobid"))
        .withColumn("UPDATE_TIME",  F.col("curr_ts"))
        .drop("original_jobid", "original_ins_time",
              "curr_jobid", "curr_ts")
    )
else:
    # First run: all records are new insertions
    df_prepared = (
        df_dedup
        .withColumn("JOBID",        F.col("curr_jobid"))
        .withColumn("INS_TIME",     F.col("curr_ts"))
        .withColumn("JOBID_UPDATE", F.col("curr_jobid"))
        .withColumn("UPDATE_TIME",  F.col("curr_ts"))
        .drop("curr_jobid", "curr_ts")
    )
```

Listing 4.15: Deduplication and audit field management in the ODS preparation step

The immutability of `JOBID` and `INS_TIME` is enforced by checking whether the ODS target table already exists at runtime. If it does, the original values are retrieved via a left join and preserved using `coalesce`. On the first pipeline execution, when no prior ODS exists, all records are treated as new insertions and both pairs of audit fields are initialized with the current run's values.

The second step exposes the preparation table as a streaming view using `readStream` with `ignoreChanges = true`, bridging the batch preparation stage to the streaming merge mechanism required by `dlt.apply_changes`. The third step creates the ODS target as a DLT streaming table. The fourth step applies the SCD1 merge:

```
dlt.apply_changes(
    target             = f"ods_{prefix}",
    source             = f"stream_prep_ods_{p_lower}",
    keys               = pk_cols,
    sequence_by        = F.col("_merge_sequence"),
    stored_as_scd_type = 1,
    except_column_list = ["_merge_sequence"]
)
```

Listing 4.16: SCD Type 1 merge via `dlt.apply_changes`

The `sequence_by` field is set to `_merge_sequence`, derived from `JOBID_UPDATE` cast to a long integer. This composite sequencing key guarantees deterministic conflict resolution by `dlt.apply_changes`, ensuring that the most recently processed version of a record always takes precedence during concurrent or replayed executions. The `except_column_list` parameter excludes `_merge_sequence` from being written to the ODS target, keeping the output schema clean.

Pipeline Orchestration and Extensibility The entire L1 layer is orchestrated via `main_l1.py`, which follows the same metadata-driven configuration pattern established in L0. Each dataset is described by a dictionary specifying its prefix, associated business rule function, and primary key columns. The orchestration loop then invokes the two factory functions in sequence:

```
datasets_l1 = [
    {
        "prefix": "<DATASET_PREFIX>",
        "rules": br.business_rules_dataset,
        "pk":    ["<key_col_1>", "<key_col_2>"]
    }
]

create_metadata_tables()

for ds in datasets_l1:
    create_ok_err_tables(
        prefix               = ds["prefix"],
        source_layer         = SOURCE_LAYER,
        target_layer         = TARGET_LAYER,
        business_rules_func  = ds["rules"],
        business_key         = ds["pk"]
    )
    create_ods_pipeline(
        prefix  = ds["prefix"],
        pk_cols = ds["pk"]
    )
```

Listing 4.17: L1 orchestration loop

Similarly to Layer L0, the L1 pipeline is orchestrated through a metadata-driven configuration and automatically generates its own execution DAG at runtime based on table dependencies and transformation logic. The resulting graph is illustrated in Figure 4.2. Together with L0, this architecture establishes a governed multi-layer data platform capable of supporting advanced analytical workloads while preserving traceability, reproducibility, and domain integrity across the full data lifecycle.

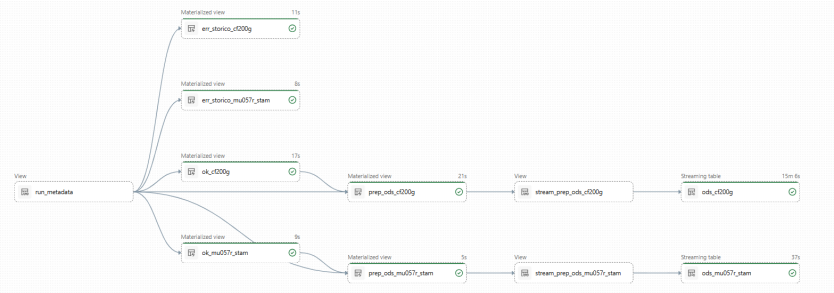

Figure 4.2: DAG representation of the Layer L1 pipeline.

4.4 Governance, Auditing, and Monitoring

A dedicated audit and governance layer was designed to ensure end-to-end observability across the multi-layer ETL pipeline. This layer is implemented through three Databricks notebooks — executed respectively at pipeline startup, at the L0-to-L1 transition, and at the conclusion of the L1 layer — and relies on two physical Delta tables: `flow_manager` and `table_manager`, stored within a dedicated audit schema in the Unity Catalog.

4.4.1 Audit Infrastructure Setup

The first notebook, executed once during environment initialisation, provisions the audit schema and the two governance tables. Rather than relying on implicit table creation, the schema and tables are declared explicitly using `CREATE TABLE IF NOT EXISTS` statements, ensuring idempotent and reproducible infrastructure setup across environments.

The `flow_manager` table tracks the lifecycle of each pipeline execution at the workflow level. Each row represents a pipeline run within a specific processing layer and carries the following attributes:

```
CREATE TABLE IF NOT EXISTS
    <catalog>.<audit_schema>.flow_manager (
    IDENTITY    STRING    NOT NULL,  -- Execution
                                     environment
                                     (e.g.,DEV, PROD)
    GRP_NAME    STRING    NOT NULL,  -- Functional domain
                                     or business area
    NUM_LEVEL   INT       NOT NULL,  -- Logical layer(0 = L0,
                                     1 = L1, 2 = L2)
    SRC_JOBID   STRING    NOT NULL,  -- Upstream execution
                                     identifier
    TRG_JOBID   STRING    NOT NULL,  -- Current execution
                                     identifier
    STATUS      INT       NOT NULL,  -- 0 = success
                                     -- 1 = in progress
                                     -- 3 = terminated
                                     with errors
    LOAD        INT       NOT NULL,  -- 0 = already
                                     consumed
                                     -- 1 = ready for
                                     consumption
                                     -- 2 = not yet ready
                                     -- 5 = final level no
                                     further consumption
```

```
25      START_DATE TIMESTAMP NOT NULL,  -- Execution start
26                                      timestamp
27      END_DATE   TIMESTAMP            -- Execution end
28                                      timestamp (NULL until
29                                      complete)
30  ) USING DELTA
```

Listing 4.18: DDL for the `flow_manager` audit table

Regarding the `STATUS` field, only the values `0` (completed successfully) and `1` (execution in progress) are operative in the current implementation. The value `3` (terminated with errors), defined in the framework specification in Chapter 3, is reserved for future extension and is not yet wired to the Databricks alerting mechanisms in this prototype deployment.

The `LOAD` field encodes pipeline readiness for downstream consumption using four discrete integer values: `0` indicates that the data produced by the layer has already been consumed by the subsequent stage; `1` indicates that the layer has completed successfully and its output is available for consumption; `2` indicates that the layer is still in progress and its output is not yet available; and `5` marks the end of the pipeline lifecycle, signalling that the final logical level has been reached and no further consumption is possible.

The `SRC_JOBID` and `TRG_JOBID` fields implement a cross-layer lineage chain. At L0, `SRC_JOBID` holds the conventional default value `19000101000000`, indicating the absence of an upstream execution. At L1, it assumes the value of the `TRG_JOBID` produced by the corresponding L0 run, enabling full cross-layer traceability via a single execution identifier.

The `table_manager` table operates at a finer granularity, recording the number of rows produced by each individual Delta Live Table within a given pipeline run:

```
1   CREATE TABLE IF NOT EXISTS
2       <catalog>.<audit_schema>.table_manager (
3       IDENTITY   STRING,
4       NUM_LEVEL  INT,
5       GRP_NAME   STRING,
6       TABLE_NAME STRING,      -- Normalised target table name
7       JOBID      STRING,      -- Execution identifier
8                                (matches TRG_JOBID)
9       NUM_ROWS   LONG,        -- Row count (LONG to prevent
                                integer overflow)
10      LOAD_DATE  TIMESTAMP    -- Timestamp of the audit
                                record insertion
11  ) USING DELTA
```

Listing 4.19: DDL for the `table_manager` audit table

The `NUM_ROWS` column is typed as `LONG` rather than `INT` to safely accommodate large-volume datasets without integer overflow risk.

Pipeline Initialization and JobID Generation The same initialization notebook is also responsible for generating the execution identifier (*jobid*) that governs the entire pipeline run. The *jobid* is derived deterministically from the current timestamp at the moment of pipeline startup, formatted as a fourteen-digit string (`YYYYMMDDHHmmss`). This format guarantees monotonic ordering across executions, enabling chronological sorting and conflict-free identification of consecutive runs.

```
run_ts     = datetime.datetime.now()
trg_jobid = run_ts.strftime("%Y%m%d%H%M%S")

spark.sql(f"""
    INSERT INTO <audit_schema>.flow_manager
    (IDENTITY, NUM_LEVEL, GRP_NAME, TRG_JOBID, SRC_JOBID,
     STATUS, LOAD, START_DATE, END_DATE)
    VALUES (
        '{IDENTITY}', 0, '{GRP_NAME}', '{trg_jobid}',
        '{DEFAULT_L0_SRC}',
        1,    -- STATUS: In progress
        2,    -- LOAD: Not yet ready
        current_timestamp(), NULL
    )
""")
```

Listing 4.20: JobID generation and pipeline initialization

The `SRC_JOBID` for the first L0 run is set to a conventional placeholder value (e.g., `"19000101000000"`), as no prior execution exists. Once the `flow_manager` record is inserted, the *jobid* is broadcast to the Delta Live Tables pipeline through a dedicated signal table:

```
meta_data = [Row(jobid=trg_jobid, run_timestamp=run_ts)]
spark.createDataFrame(meta_data) \
     .write.format("delta").mode("overwrite") \
     .saveAsTable(f"<audit_schema>.run_metadata_signal")
```

Listing 4.21: Execution signal propagation via `run_metadata_signal`

The `run_metadata_signal` table is written in overwrite mode, ensuring that it always contains exactly one row — the current execution context. This table is subsequently consumed by both the L0 and L1 DLT pipelines through their respective `run_metadata` views, as described in earlier sections. Databricks natively manages the metadata of all DLT-registered tables through its internal metastore and Unity Catalog integration; the `run_metadata_signal` table complements this

infrastructure by providing a lightweight, pipeline-readable execution anchor that is not directly exposed through the DLT framework.

4.4.2 L0-to-L1 Transition Notebook

Upon successful completion of the L0 pipeline, a transition notebook is executed as the subsequent task in the Databricks workflow. This notebook performs three operations: it extracts row-level metrics from the L0 DLT event log and writes them to `table_manager`; it closes the L0 record in `flow_manager` by setting `STATUS = 0` and `LOAD = 1`; and it opens a new L1 record, linking it to the current *jobid* to establish cross-layer lineage.

Metadata extraction relies on the native `event_log()` function provided by Databricks for Delta Live Tables. This function exposes structured execution events, including `flow_progress` records containing output row-count metrics. The most recent `update_id` is first resolved to isolate the current pipeline execution. Subsequently, `flow_progress` events associated with that update are filtered and parsed to extract per-table row counts:

```
latest_update_id = spark.sql(f"""
    SELECT origin.update_id FROM event_log('{pipeline_id}')
    WHERE event_type = 'create_update'
    ORDER BY timestamp DESC LIMIT 1
""").collect()[0]['update_id']

json_schema = (
    "flow_progress STRUCT"
    "<metrics STRUCT<num_output_rows LONG>>"
)

df_metrics = (
    spark.sql(
        f"SELECT origin.flow_name, timestamp, details "
        f"FROM event_log('{pipeline_id}')"
    )
    .filter(F.col("origin.update_id") == latest_update_id)
    .filter(F.col("event_type") == "flow_progress")
    .withColumn("parsed",
                F.from_json(F.col("details"), json_schema))
    .withColumn("num_rows",
                F.col("parsed.flow_progress.metrics"
                      ".num_output_rows"))
    .filter(F.col("num_rows").isNotNull())
)
```

Listing 4.22: DLT metric extraction from the pipeline event log

Since a single DLT flow may emit multiple `flow_progress` events during execution, a window function deduplicates the results by retaining only the most recent event per table, identified by descending timestamp. Table names are normalized by extracting the last segment of the fully qualified name, stripping catalog and schema prefixes. The resulting records are appended to `table_manager` in Delta append mode.

Following metric consolidation, the L0 lifecycle record is closed and the L1 record is opened in `flow_manager`:

```
-- Close L0 execution record
UPDATE <audit_schema>.flow_manager
SET STATUS = 0, LOAD = 1, END_DATE = current_timestamp()
WHERE TRG_JOBID = '<trg_jobid>'
  AND NUM_LEVEL = 0
  AND GRP_NAME  = '<grp_name>';

-- Open L1 execution record, linking SRC_JOBID to L0
INSERT INTO <audit_schema>.flow_manager
(IDENTITY, GRP_NAME, NUM_LEVEL, SRC_JOBID, TRG_JOBID,
 STATUS, LOAD, START_DATE, END_DATE)
VALUES ('<identity>', '<grp_name>', 1,
        '<trg_jobid>', '<trg_jobid>',
        1, 2, current_timestamp(), NULL);
```

Listing 4.23: Flow Manager state transition at the L0-to-L1 boundary

The choice to use the same value for both `SRC_JOBID` and `TRG_JOBID` in the L1 record is intentional: since both layers share the same *jobid* generated at initialization, this design encodes the within-run dependency between L0 and L1 while preserving the lineage link. The `LOAD` flag is set to `2` (not yet ready) at opening and will be updated to `5` (complete) upon L1 closure.

4.4.3 L1 Closure Notebook

The final notebook in the workflow is executed after the L1 DLT pipelines complete. It mirrors the metric extraction logic of the transition notebook but operates across multiple L1 pipeline identifiers — one per logical dataset group — invoking the extraction function independently for each and appending all results to `table_manager`. Once all metrics are recorded, the L1 record in `flow_manager` is closed:

```
UPDATE <audit_schema>.flow_manager
SET STATUS = 0, LOAD = 5, END_DATE = current_timestamp()
WHERE TRG_JOBID = '<trg_jobid>'
  AND NUM_LEVEL = 1
```

```
5   AND GRP_NAME   = '<grp_name>';
```

Listing 4.24: L1 Flow Manager closure

The `LOAD = 5` value marks the workflow as fully complete and ready for downstream consumption, distinguishing it from intermediate states such as `LOAD = 1` (L0 complete, L1 pending).

Design Rationale The three-notebook audit architecture was designed to achieve a clean separation between pipeline execution and governance concerns. By externalizing audit logic into dedicated notebooks executed as distinct workflow tasks, the DLT pipelines themselves remain focused exclusively on data transformation. The `flow_manager` table provides a persistent, queryable record of workflow state transitions, enabling monitoring and alerting at the orchestration level. The `table_manager` table complements this with fine-grained row-count telemetry per table per run, supporting data volume trend analysis and anomaly detection. Together, these two tables form the observability backbone of the platform, covering the full execution lifecycle from pipeline initialization through cross-layer handoff to final closure.

4.5 Pipeline Orchestration and Execution

4.5.1 Databricks Jobs Configuration

Databricks Jobs [11] constitute the native orchestration mechanism within the Databricks platform, enabling the definition, scheduling, and monitoring of complex multi-step workflows. A Job is a configurable execution unit composed of one or more tasks, each of which may represent a Databricks notebook, a Delta Live Tables pipeline, a Python script, or an arbitrary compute workload. Tasks within a Job can be arranged with explicit dependency relationships, forming a directed acyclic graph that governs execution order and parallelism.

Jobs support multiple triggering modalities: they can be executed on demand, scheduled via cron expressions, or triggered programmatically through the Databricks REST API. For production environments, cron-based scheduling ensures that the ETL pipeline executes automatically at predefined intervals — for instance, nightly or at the beginning of each business day — without requiring manual intervention.

A key operational feature is the integrated alerting mechanism: Jobs can be configured to dispatch email notifications to designated recipients upon task failure, successful completion, or execution timeout. This capability provides immediate visibility into pipeline health and enables rapid incident response without relying on external monitoring infrastructure. Furthermore, the Databricks Jobs UI

exposes detailed execution telemetry for each run, including per-task start and end timestamps, total elapsed time, and resource utilization, enabling systematic performance tracking and capacity planning over time.

Data Lineage An additional observability capability provided by Databricks through Unity Catalog is data lineage tracking. Lineage captures the full provenance graph of data assets: for each table registered in the catalog, the platform automatically traces which upstream sources contributed to its content and which downstream tables or queries consume it. This graph is maintained at the column level in Unity Catalog-enabled workspaces, allowing data engineers and governance teams to assess the impact of schema changes, trace the origin of data quality issues, and demonstrate regulatory compliance by documenting how sensitive data flows across the platform. Within the context of the present architecture, lineage provides end-to-end visibility from raw landing zone files through staging, cleansing, CDC, validation, and ODS consolidation, making the entire transformation chain auditable without manual documentation effort.

Job Structure and Task Dependencies The ETL workflow is implemented as a single Databricks Job composed of six tasks. The task structure reflects the logical organization of the pipeline, separating Layer L0 and Layer L1 processing while enforcing the required referential dependencies between datasets.

The first task, **audit_start**, executes the initialization notebook described in Section 4.4.1. This notebook generates the *jobid* for the current run, registers the L0 execution record in the `flow_manager` table, and writes the execution signal to `run_metadata_signal`. All downstream tasks depend on the successful completion of this step, as the *jobid* it produces constitutes the shared execution context for the entire workflow.

The second task triggers the **L0 pipeline**, which ingests, cleanses, and applies CDC logic across all configured datasets simultaneously. Because Delta Live Tables internally resolves inter-table dependencies through lineage analysis and constructs its own execution DAG, all staging, cleansing, and CDC tables for every dataset are materialized within a single pipeline update. No external task-level sequencing is required within L0 itself.

Upon successful completion of the L0 pipeline, the third task executes the **audit_middle** notebook, which extracts row-level metrics from the L0 event log, consolidates them into the `table_manager` table, closes the L0 record in `flow_manager`, and opens the corresponding L1 record — establishing the cross-layer lineage link described in Section 4.4.2.

The fourth and fifth tasks trigger the two **L1 pipelines** — one dedicated to registry datasets and one to transactional datasets. These two pipelines are executed sequentially rather than in parallel: the registry pipeline runs first, and

the transactional pipeline is configured with an explicit dependency on it. This ordering is deliberate. Transactional datasets contain foreign key references to entities defined in the registry domain; processing registry data first ensures that referential integrity can be verified during L1 validation without risk of missing master data. Executing both pipelines within the same Job also guarantees that both operate under the same *jobid*, preserving execution cohesion across dataset groups.

The sixth and final task executes the **audit_end** notebook, which extracts and consolidates row-level metrics for both L1 pipelines into `table_manager` and closes the L1 record in `flow_manager` with `STATUS = 0` and `LOAD = 5`, marking the workflow as fully complete and ready for downstream consumption.

The complete task dependency chain is illustrated in Figure 4.3.

Figure 4.3: Structural configuration of the Databricks Job.

This sequential structure ensures that each governance checkpoint is executed only after its corresponding pipeline stage has successfully concluded, preventing partial or inconsistent audit records. Should any task fail, Databricks halts execution of all downstream dependents and dispatches a failure notification to the configured recipients, allowing the pipeline state to be inspected and remediated before the next scheduled run.

The empirical execution behavior observed across multiple pipeline runs is analysed in Chapter 6. The data assets produced by this pipeline — in particular the validated and consolidated ODS tables — constitute the analytical foundation upon which the machine learning and forecasting applications described in Chapter 5 are built.

Chapter 5

Artificial Intelligence Applications

5.1 Data Governance for Analytics and Data Science

Before describing the modelling work, it is necessary to address the governance framework that regulates access to the data platform for analytical purposes. In a production enterprise context, a data platform serves multiple teams with distinct roles and responsibilities, each requiring access to different subsets of data. Establishing a principled access control strategy is therefore a prerequisite for compliant and secure data science operations.

5.1.1 Access Control and User Groups

Databricks, when integrated with Unity Catalog, provides a centralized identity and access management framework that operates at the level of catalogs, schemas, tables, and individual columns. Permissions are managed through a role-based model in which users are assigned to groups, and access rights are granted to groups rather than to individuals. This approach reduces administrative overhead, ensures consistency, and simplifies auditing by making permission assignments explicit and queryable.

To explore this capability in a realistic scenario, a dedicated user group named `Data Science` was provisioned within the Databricks workspace to simulate the organizational separation between a data integration team and a data science team. In an enterprise setting, the data integration team — responsible for designing and maintaining the ETL pipelines — requires broad access to the data platform, including raw, intermediate, and operational tables, as well as metadata and audit

infrastructure. The data science team, by contrast, requires access only to the specific tables needed for model development, and only to the columns that carry analytical signal.

The first step in operationalizing this separation was the creation of a dedicated schema within the existing catalog, reserved exclusively for machine learning and analytical activities:

```
CREATE SCHEMA IF NOT EXISTS <catalog>.ml_analysis
COMMENT 'Schema dedicated to ML and Analytics activities';
```

Listing 5.1: Creation of a dedicated schema for machine learning workloads

Isolating analytical workloads within a dedicated schema serves multiple purposes. It provides a clear organizational boundary between operational and analytical data assets, simplifies permission management by allowing grants to be defined at the schema level, and ensures that any views or feature tables created by the data science team do not interfere with the production ODS tables.

Access for the `Data Science` group was then configured through a layered permission structure, proceeding from the catalog level down to the schema level:

```
-- Grant catalog-level visibility
GRANT USE CATALOG ON CATALOG <catalog> TO `Data Science`;

-- Grant schema-level visibility
GRANT USE SCHEMA ON SCHEMA <catalog>.ml_analysis
    TO `Data Science`;

-- Grant read access to all current and future tables
GRANT SELECT ON SCHEMA <catalog>.ml_analysis TO `Data
   Science`;
```

Listing 5.2: Layered permission grants for the Data Science group

This layered approach reflects the hierarchical permission model enforced by Unity Catalog: a user must hold `USE CATALOG` before being able to interact with any schema within it, and `USE SCHEMA` before being able to access any table within that schema. Granting `SELECT` at the schema level — rather than on individual tables — ensures that newly created analytical tables are automatically accessible to the group without requiring additional permission updates, reducing administrative friction while maintaining control.

This distinction has concrete privacy and compliance implications in the context of financial data. The ODS tables produced by the L1 pipeline may contain personal identifiers — such as names, contact details, or fiscal codes — that are necessary for entity resolution and data integration purposes but carry no predictive value for machine learning models. Granting the data science group unrestricted access

to such attributes would constitute unnecessary exposure of personal data, in tension with the data minimisation principle as codified in Article 5(1)(c) of the General Data Protection Regulation [23]. By exposing to the `Data Science` group only a dedicated table within the `ml_analysis` schema — populated exclusively with the columns selected for their analytical relevance, namely financial metrics, behavioral indicators, risk scores, and temporal attributes — the platform enforces least-privilege access without requiring physical data duplication. In Databricks Unity Catalog, this granularity is further reinforced through column-level access controls and dynamic data masking, which allows sensitive columns to be exposed in redacted form to groups that do not hold full read privileges.

5.1.2 Historical Snapshot Management for Machine Learning

The Need for Temporal Snapshots

A critical consideration in connecting the ETL pipeline to the machine learning layer concerns the temporal structure of the data required for modelling. The ODS layer implements a Slowly Changing Dimension Type 1 strategy: when a record is updated, its attributes are overwritten with the most recent values, meaning that historical states are not preserved within the ODS itself. This property is incompatible with machine learning tasks that require observations across multiple time periods.

In this project, the ETL pipeline was executed twice to simulate two monthly data delivery cycles. After the second run, the ODS contained only the January state, with no accessible record of December values. Had the ML feature table been re-populated directly from the ODS at that point, the December snapshot — and with it the ability to derive temporal features such as changes in overdue instalments or days past due — would have been irretrievably lost. Using the upstream `stg_clean` staging tables as an alternative would have provided both snapshots, but at the cost of bypassing the L1 validation layer entirely, exposing the model to records that have not passed business rule enforcement and breaking the semantic continuity between the ETL and ML layers.

Accumulation Table Strategy

To resolve this tension, the ML feature table within the `ml_analysis` schema was designed as an *accumulation table* — a table whose contents grow monotonically over time through the append-only insertion of new snapshots rather than the overwriting of existing records.

The schema retains only attributes with analytical value — financial indicators such as outstanding balance, overdue capital, and net risk, contract metadata,

and risk classification fields — deliberately excluding all personal identifiers and operationally sensitive columns present in the ODS, enforcing data minimisation by design. Two additional columns are introduced: `snapshot_periodo`, a human-readable label derived from the reference date (e.g., *December* or *January*), and `processing_timestamp`, which records when each record was materialised into the analytical layer.

The population logic operates in two steps after each pipeline execution. First, the current ODS state is read, transformed, and labelled with the reference period. Second, the transformed records are merged into the history table using a composite key formed by the entity identifier (`ndg`), the contract identifier (`numero_rapporto`), and the reference date (`data_riferimento`). This key guarantees that December and January records for the same entity coexist as distinct rows without overwriting each other, and makes the operation fully idempotent: re-executing the accumulation logic for a period that has already been processed updates existing rows in place rather than duplicating them.

After the two pipeline executions simulated in this project, the history table contains two rows per entity — one labelled *December* and one labelled *January* — which the machine learning layer can join to derive temporal features capturing the within-entity evolution between the two reference periods. It should be noted that the write strategy adopted here — a MERGE keyed on the composite identifier — differs deliberately from the simpler `overwrite` mode used for the scoring output table described in Section 5.2.6. The accumulation table is designed as a long-term feature store whose history must be preserved across runs, whereas the scoring output represents the most recent portfolio assessment and is replaced at each monthly execution.

5.1.3 Data Sharing and Security Constraints

Beyond internal access control, the Unity Catalog framework supports governed data sharing through Delta Sharing [31], an open protocol that enables organizations to share live Delta Lake data with external recipients without data movement or replication. In the context of the present platform, Delta Sharing could be leveraged to expose curated ODS views to external analytical teams or partner organizations while preserving full auditability of access events.

Within the internal platform, security constraints are enforced at multiple levels. At the catalog and schema level, access grants determine which groups can discover and enumerate data assets. At the table level, read and write privileges are assigned independently, ensuring that analytical users cannot inadvertently modify production ODS tables. At the column level, sensitive attributes are either excluded from the data science group's grants — by surfacing only pre-filtered content within the `ml_analysis` schema — or masked through Unity Catalog's

native masking policies.

Collectively, these access control and data sharing mechanisms ensure that the transition from a governed ETL platform to an analytical consumption layer does not introduce uncontrolled data exposure. By treating data governance not as an afterthought but as a structural component of the platform design, the architecture supports compliant, auditable, and secure data science operations from the point of data ingestion through to model development. The machine learning pipeline described in the following section operates entirely within this governed framework, consuming exclusively the validated and access-controlled outputs of the accumulation layer.

5.2 Machine Learning for Financial Risk Assessment

The growing complexity of credit markets and the increasing volume of financial data available to lending institutions have created both the need and the opportunity to apply machine learning techniques to credit risk assessment. Traditional statistical approaches, such as logistic regression scoring models [37], while interpretable and well-established in regulatory frameworks, are often limited in their ability to capture non-linear relationships between borrower characteristics and default probability. This section describes the development of a machine learning pipeline designed to predict short-term credit deterioration within the loan portfolio of a financial intermediary, with the objective of producing calibrated probability of default (PD) estimates at the individual loan level on a monthly basis. The pipeline consumes the validated and governed feature data produced by the ODS layer of the ETL framework described in Chapter 4, accessed through the accumulation table structure introduced in Section 5.1.

The system is designed to operate as an early warning tool: given a snapshot of the portfolio at the end of a reference month, the model estimates the probability that each performing loan will exhibit signs of deterioration by the end of the following month. The output is intended to support credit officers in prioritising monitoring activities and allocating remediation resources before formal non-performing loan (NPL) classifications are triggered. The pipeline operates entirely within the governed infrastructure described in Section 5.1.

5.2.1 Problem Definition and Target Variable

The starting point of any supervised learning problem in credit risk is the precise definition of the event to be predicted — the so-called *target variable*. This choice is not merely technical: it encodes the institution's definition of credit deterioration

and directly determines what behaviour the model will learn to anticipate. An imprecise or overly broad target can lead to a model that is statistically well-fitted but operationally meaningless; conversely, an excessively restrictive target may fail to capture early warning signals that precede formal default classifications.

The prediction task is framed as a binary classification problem over a one-month horizon. The dataset is constructed by merging two monthly snapshots of the loan management system: the December snapshot, which provides the feature set, and the January snapshot, which provides the ground truth for the target variable. Only loans with an active status (`stato_del_rapporto_mu = "EROGATO"`) in both periods are retained, ensuring that the analysis focuses exclusively on performing exposures at the observation date. The merge is performed with explicit column renaming rather than relying on automatic suffix generation, guaranteeing unambiguous traceability of each variable to its reference period:

```
df_old_renamed = df_old[cols_old].rename(
    columns={c: f"{c}_old" for c in cols_to_rename}
)
df_new_renamed = df_new[cols_new].rename(columns={
    "rate_scadute_non_pagate":     "rate_scadute_non_pagate_new",
    "giorni_di_insoluto":          "giorni_di_insoluto_new",
    "classificazione_rapporto_mu": "
   classificazione_rapporto_mu_new"
})
df_merged = df_new_renamed.merge(
    df_old_renamed, on="numero_rapporto", how="inner"
)
```

Listing 5.3: Merge of December and January snapshots with explicit renaming

The use of an inner join is deliberate: loans that do not appear in both snapshots — whether due to early repayment, transfer, or data quality issues — are excluded from the training population, ensuring the model is trained exclusively on loans for which a complete observation window is available.

A loan is labelled as *deteriorated* (target = 1) if at least one of three conditions is observed between December and January. The first captures the *emergence of overdue instalments*: a loan transitions from zero unpaid instalments to a strictly positive number — the most direct signal of payment stress, reflecting a borrower's first failure to meet a contractual obligation. The second captures *accelerating delinquency*: days in arrears increase by more than fifteen days, a threshold calibrated to filter out minor administrative processing delays while flagging genuinely worsening arrears. The third captures *formal rating migration*: a loan previously classified as *bonis* transitions to any inferior classification, the most severe signal as it reflects a credit officer's judgment that reclassification is warranted. These three conditions are combined with a logical OR:

```
df_merged["target"] = np.where(
    (
        (df_merged["rate_scadute_non_pagate_old"] == 0) &
        (df_merged["rate_scadute_non_pagate_new"] > 0)
    ) |
    (
        (df_merged["giorni_di_insoluto_new"] -
         df_merged["giorni_di_insoluto_old"]) > 15
    ) |
    (
        df_merged["classificazione_rapporto_mu_old"]
            .str.contains("BONIS", na=False) &
       ~df_merged["classificazione_rapporto_mu_new"]
            .str.contains("BONIS", na=False)
    ),
    1, 0
)
```

Listing 5.4: Definition of the binary target variable

This composite definition is deliberately conservative: it prioritises recall over precision at the target construction stage, encompassing a broad set of early-stage deterioration signals. It is preferable to include borderline cases in the positive class and allow the model to learn their distinguishing characteristics, rather than restricting the target to only the most severe events, which would reduce the training signal available for early warning purposes.

A critical methodological consideration concerns *target leakage* [38]. All features are drawn exclusively from the December snapshot (variables suffixed `_old`), while the target is constructed from the January snapshot (variables suffixed `_new`). This temporal separation is strictly enforced throughout the pipeline. In practice, particular care must be taken with variables recorded at month-end boundaries—for instance, overdue amounts computed on 31 December that may already incorporate early-January accruals depending on the system's accounting logic. Such cases should therefore be reviewed with the data engineering team before each model update.

The resulting dataset exhibits a marked class imbalance, consistent with the expected prevalence of early-stage deterioration in a performing loan portfolio: approximately 8.02% of observations are labelled as deteriorated, while 91.98% correspond to stable exposures. This 1:11 ratio is consistent with the institutional context and does not suggest a systematic error in the target definition; however, it necessitates explicit handling at the modelling stage to prevent the classifier from collapsing to the trivial majority-class solution. This issue is addressed through the `scale_pos_weight` mechanism described in Section 5.2.3.

5.2.2 Feature Engineering and Relevance Analysis

Construction of Derived Features

The raw variables available in the source system carry meaningful financial information but are difficult to compare across loans of different sizes in their absolute form. A loan with an outstanding balance of 500,000€ is not inherently riskier than one with a balance of 50,000€; what matters is the balance *relative* to the original exposure, the credit limit, or the contractual repayment schedule. The feature engineering stage therefore focuses on *normalised ratios* that capture the financial structure of each loan in a size-invariant manner, while retaining key absolute measures of delinquency intensity.

All ratios are computed through a numerically robust helper function that guards against division by zero — a frequent occurrence where some fields are zero for structural reasons such as revolving facilities with no fixed amortisation schedule:

```
1 def safe_ratio(num, den, upper_clip=None):
2     ratio = (num / den.replace(0, np.nan)).fillna(0)
3     return ratio.clip(upper=upper_clip) if upper_clip else ratio
```

Listing 5.5: Safe ratio computation with optional upper clipping

When the denominator is zero the result is set to zero rather than infinity or NaN, and an optional upper clip prevents extreme values from distorting the feature distribution. Clipping thresholds are set conservatively — slightly above the theoretical maximum of each ratio — to absorb legitimate edge cases such as a balance marginally exceeding original capital due to accrued interest.

Four ratio features are constructed. The *instalment payment rate* (`perc_rate_pagate`), clipped at 1.0, measures the fraction of total scheduled instalments paid to date, providing a longitudinal view of repayment discipline across the life of the loan. The *residual debt ratio* (`perc_debito_residuo`), clipped at 1.5, is the current outstanding balance divided by the initial loan amount, capturing the stage of amortisation: loans that have barely reduced their principal carry higher short-term credit risk. The *credit facility utilisation rate* (`perc_utilizzo_fido`), clipped at 1.5, expresses outstanding balance as a fraction of the current credit limit; high utilisation is a well-established signal of financial stress in SME lending, indicating limited residual liquidity margin. The *instalment-to-balance ratio* (`rata_su_saldo`), clipped at 0.5, approximates repayment intensity and is inversely related to residual maturity.

Three delinquency variables are derived directly from the December snapshot. `flag_gia_scaduti` is 1 if the borrower had at least one unpaid overdue instalment at the observation date. `flag_gia_insoluto` is 1 if days in arrears were strictly positive. These binary flags capture the *presence* of pre-existing stress. Complementing them, `giorni_di_insoluto_old` is retained as a continuous measure of

delinquency *severity*: a borrower one day past due presents a materially different risk profile from one forty-five days past due, and retaining both the flag and the continuous variable allows the model to exploit this distinction. Missing values are conservatively imputed to zero, reflecting the absence of delinquency.

Two further variables are included without transformation: `fatturato_mu_old`, the borrower's annual turnover, which serves as a proxy for business size and debt-servicing capacity; and the categorical variable `flag_forborne_old`, indicating whether the loan was subject to a forbearance measure. Rather than one-hot encoding this variable — which would increase dimensionality and lose the natural ordinality of the categories — it is cast to the `category` dtype natively supported by XGBoost 2.0 [39], which handles categorical splits internally through an optimised partitioning algorithm:

```
cat_cols = ["flag_forborne_old"]
for col in cat_cols:
    df_ml[col] = df_ml[col].astype("category")
```

Listing 5.6: Native categorical encoding for XGBoost

The final feature set comprises nine variables, summarised in Table 5.1.

Table 5.1: Summary of features used in the model

Feature	Type	Description
`flag_gia_scaduti`	Binary	1 if overdue instalments > 0 at December
`flag_gia_insoluto`	Binary	1 if days in arrears > 0 at December
`giorni_di_insoluto_old`	Numeric	Days in arrears at December (0 if missing)
`perc_rate_pagate`	Numeric	Fraction of scheduled instalments paid
`perc_debito_residuo`	Numeric	Outstanding balance / initial capital
`perc_utilizzo_fido`	Numeric	Outstanding balance / credit limit
`rata_su_saldo`	Numeric	Contractual instalment / outstanding balance
`flag_forborne_old`	Categorical	Whether loan is under forbearance
`fatturato_mu_old`	Numeric	Annual turnover (€)

Class Distribution

As shown in Figure 5.1, the target variable exhibits a 1:11 imbalance consistent with the characteristics of a performing portfolio monitored at monthly frequency.

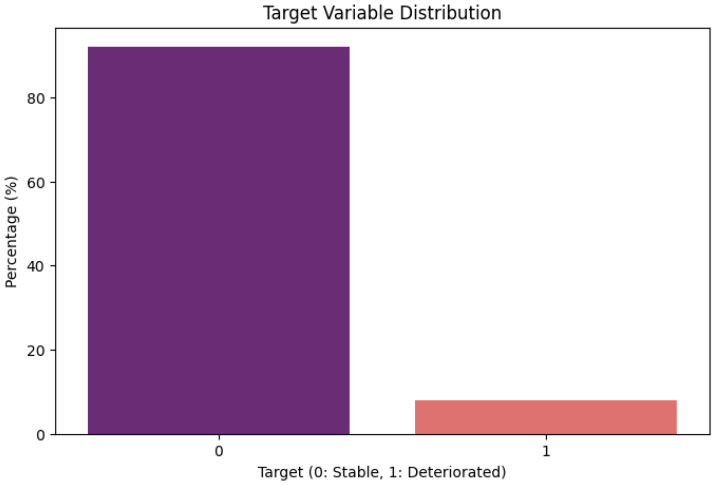

Figure 5.1: Distribution of the target variable.

Feature Relevance Analysis via Mutual Information

Prior to model training, the statistical relevance of each feature is assessed using *mutual information* (MI) [40], a non-parametric measure of statistical dependence that captures both linear and non-linear associations. Unlike the Pearson correlation coefficient, MI quantifies the reduction in uncertainty about the target achieved by observing each feature without assuming any functional form, making it particularly suited for credit risk applications where relationships between financial ratios and default probability are known to be non-linear. Categorical features are encoded as integer codes for this step only, as required by scikit-learn's `mutual_info_classif`; this encoding does not affect the feature representation used during training.

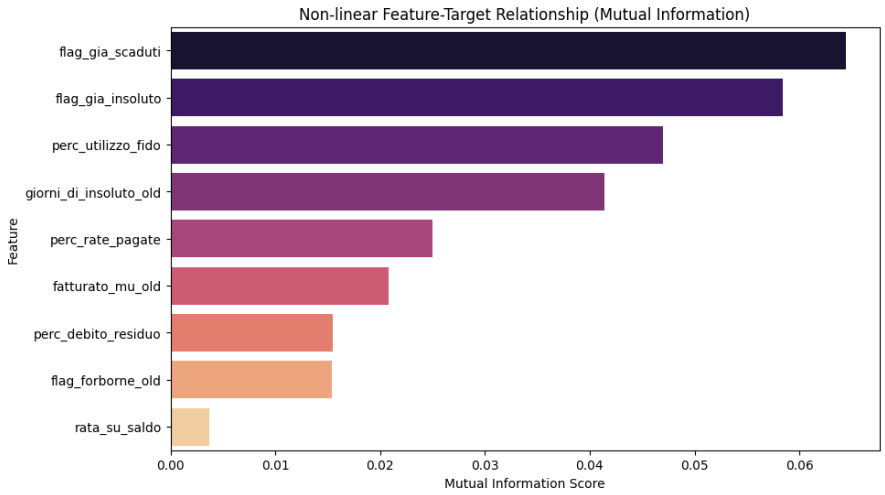

Figure 5.2: Mutual information scores between each feature and the target variable, capturing both linear and non-linear associations.

The results reveal a clear and economically interpretable hierarchy. The two binary delinquency flags dominate the ranking (MI 0.064 and 0.058), consistent with the extensive credit risk literature documenting the primacy of recent payment stress as a default predictor. The credit facility utilisation rate ranks third (MI 0.047), reflecting the well-established relationship between high balance-to-limit ratios and financial distress in SME lending. The continuous `giorni_di_insoluto_old` ranks fourth (MI 0.041), providing complementary severity information: as discussed above, the presence and the magnitude of arrears are distinct signals, and both are justified on statistical and economic grounds. The instalment payment rate (MI 0.025) captures medium-term repayment discipline, complementing the short-term delinquency indicators. Annual turnover (MI 0.021) proxies debt-servicing capacity, while the residual debt ratio and forbearance flag show moderate marginal relevance (0.015–0.016). The instalment-to-balance ratio exhibits the lowest score (0.004).

Table 5.2: Mutual information scores for all features, sorted by descending relevance

Feature	**MI Score**
`flag_gia_scaduti`	0.0644
`flag_gia_insoluto`	0.0584
`perc_utilizzo_fido`	0.0470
`giorni_di_insoluto_old`	0.0414
`perc_rate_pagate`	0.0250
`fatturato_mu_old`	0.0208
`perc_debito_residuo`	0.0155
`flag_forborne_old`	0.0154
`rata_su_saldo`	0.0037

No feature is excluded on the sole basis of its MI score. Mutual information measures marginal relevance in isolation and does not capture the complementary information that lower-ranked features may contribute in a multivariate context. Feature selection is therefore delegated to the gradient boosting algorithm, which assigns splits according to an information gain criterion that naturally downweights uninformative variables by relegating them to fewer and shallower tree nodes.

5.2.3 Model Selection and Training Process

Choice of Algorithm

Gradient boosted decision trees are a natural choice for this task. Unlike linear classifiers, tree-based ensemble methods approximate arbitrary non-linear decision boundaries and capture high-order interaction effects without explicit specification — particularly valuable in credit risk, where the joint effect of multiple stress signals is typically more informative than any individual signal in isolation. XGBoost [39] is selected among gradient boosting implementations for its computational efficiency on tabular data, native support for categorical features in version 2.0, and its extensive adoption in the financial machine learning literature. The histogram-based tree construction algorithm (`tree_method="hist"`) is used throughout, grouping continuous values into discrete bins to reduce training time with negligible impact on predictive accuracy.

Dataset Partitioning

The dataset is partitioned into three non-overlapping subsets using a stratified 60%–20%–20% scheme in order to preserve the class distribution across splits:

```
1 X_train, X_temp, y_train, y_temp = train_test_split(
2     X, y, test_size=0.4, stratify=y, random_state=42
3 )
4 X_val, X_test, y_val, y_test = train_test_split(
5     X_temp, y_temp, test_size=0.5, stratify=y_temp, random_state
    =42
6 )
```

Listing 5.7: Stratified three-way split of the dataset

This three-way partitioning is essential to prevent data leakage and to ensure a statistically reliable assessment of model performance. The **training set** is used exclusively for model fitting. The **validation set** serves a dual and sequential role: first, it guides hyperparameter optimisation within the Optuna study, enabling the selection of the best-performing configuration; second, it provides the sample on which the Platt scaling calibrator is fitted, using the probability scores produced by the base classifier trained solely on the training partition. It should be acknowledged

that the reuse of the same partition for both hyperparameter selection and calibrator fitting introduces a mild optimistic bias in the calibration estimates: since the validation set has already influenced model selection, it is not strictly independent for the subsequent calibration step. In the context of this study, where the limited dataset size prevents the allocation of a separate calibration partition without materially reducing the training population, this trade-off is accepted as a practical constraint; calibration quality is assessed empirically on the held-out test set via a reliability diagram, as reported in Chapter 6. The **test set** remains strictly held out throughout the entire modelling process and is accessed only once, after all modelling decisions have been finalised, to produce an unbiased estimate of generalisation performance. Stratification is applied at both splitting stages to ensure that the minority class is proportionally represented in each partition — a particularly important consideration given the small absolute number of positive cases.

Class Imbalance Handling

The 1:11 class ratio is addressed through the `scale_pos_weight` parameter of XGBoost, set to the ratio of negative to positive instances in the training partition:

```
scale_pos = (y_train == 0).sum() / max((y_train == 1).sum(), 1)
```

Listing 5.8: Automatic computation of class weight for imbalance correction

This assigns a multiplicative weight to positive-class observations during loss computation, equivalent to oversampling the minority class by the same factor in expectation but without the increased computational cost and overfitting risk associated with resampling methods such as SMOTE [41]. The classifier is thereby discouraged from collapsing to the trivial solution of predicting the majority class for all observations — a failure mode that would yield high accuracy but zero recall on the class of interest.

Hyperparameter Optimisation via Bayesian Search

The hyperparameter configuration is determined through Bayesian optimisation using the Optuna framework [42] with the Tree-structured Parzen Estimator (TPE) sampler [43]. Bayesian optimisation constructs a probabilistic model of the objective function—in this case the validation AUC—and uses it to concentrate the search budget in promising regions of the hyperparameter space. This strategy is substantially more efficient than grid or random search when individual training runs are computationally costly. The TPE sampler models the distribution of promising and non-promising hyperparameter configurations separately, allowing the search process to focus on regions of the parameter space that are more likely to yield performance improvements.

```
def objective(trial):
    params = {
        "booster":            "gbtree",
        "tree_method":        "hist",
        "enable_categorical": True,
        "scale_pos_weight":   scale_pos,
        "n_estimators":  trial.suggest_int("n_estimators", 150,
    600),
        "max_depth":     trial.suggest_int("max_depth", 3, 6),
        "learning_rate": trial.suggest_float(
                             "learning_rate", 0.01, 0.2, log=True)
    ,
        "subsample":     trial.suggest_float("subsample", 0.6,
    1.0),
        "colsample_bytree": trial.suggest_float(
                                "colsample_bytree", 0.6, 1.0),
        "min_child_weight": trial.suggest_int(
                                "min_child_weight", 3, 10),
    }
    clf = XGBClassifier(**params)
    clf.fit(X_train, y_train, verbose=False)
    return roc_auc_score(y_val, clf.predict_proba(X_val)[:, 1])

study = optuna.create_study(
    direction="maximize", sampler=TPESampler(seed=42)
)
study.optimize(objective, n_trials=40)
```

Listing 5.9: Optuna objective function with TPE search over six key hyperparameters

The objective function evaluates model performance on a held-out validation set using the ROC-AUC metric, preserving the statistical independence between the training and validation samples.

The search space bounds are motivated by domain knowledge and regularisation considerations. `n_estimators` is searched in $[150, 600]$ to allow sufficient ensemble capacity while keeping the computational cost of each trial manageable; the number of boosting rounds is treated as a tunable hyperparameter rather than relying on early stopping, as no internal evaluation set is passed to the fitting procedure. `max_depth` is capped at 6 to limit model complexity on a relatively small training population, thereby reducing the risk of overfitting on the minority class. `learning_rate` is searched on a logarithmic scale in $[0.01, 0.2]$, as its effect on convergence is approximately multiplicative.

The subsampling rates `subsample` and `colsample_bytree`, both constrained to $[0.6, 1.0]$, introduce stochasticity at the row and feature level respectively, acting as implicit regularisation while preserving sufficient information per tree. The

`min_child_weight` parameter, searched between 3 and 10, prevents splits supported by very few minority-class observations, reducing model variance at the cost of a small increase in bias. The optimal configuration identified after 40 trials is reported in Table 6.2 in Chapter 6.

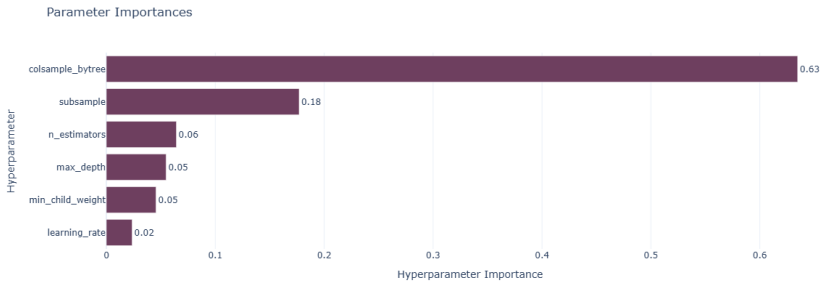

Figure 5.3: Hyperparameter importance estimated by Optuna

Final Model Training

Once the optimal hyperparameter configuration is identified, the final model is retrained on the union of the training and validation sets. This is standard practice in supervised learning pipelines: the validation set has already fulfilled its role for hyperparameter selection, while the Platt scaling calibrator has been fitted on the intermediate classifier trained exclusively on the training partition. Incorporating the validation observations into the final training data therefore introduces no additional information leakage while increasing the effective training sample size—a meaningful gain given the limited size of the dataset and the scarcity of minority-class examples.

```
X_train_full = pd.concat([X_train, X_val])
y_train_full = pd.concat([y_train, y_val])

xgb_model = XGBClassifier(**best_params)
xgb_model.fit(X_train_full, y_train_full)
```

Listing 5.10: Final model training on the combined train and validation sets

Feature importance is then extracted using the *gain* criterion—the average improvement in loss brought by splits on each feature across all trees. This measure is preferred over simple split counts because it is less susceptible to bias towards high-cardinality continuous variables and more directly reflects predictive contribution.

The gain-based ranking complements the earlier mutual information analysis. While mutual information measures marginal statistical association between each

feature and the target in isolation, gain importance reflects the contribution of each variable within the joint multivariate model learned by the gradient boosting ensemble. The empirical results from both analyses are discussed in Chapter 6.

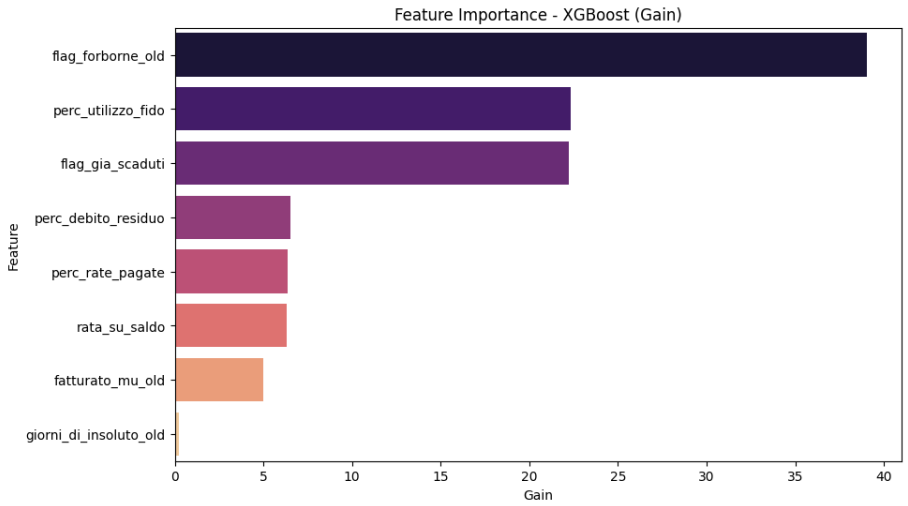

Figure 5.4: Feature importance based on the average gain criterion from the final XGBoost model.

5.2.4 Probability Calibration and Model Evaluation

Probability Calibration via Platt Scaling

Gradient boosted classifiers such as XGBoost are primarily optimised for discriminative performance (e.g., log-loss or AUC) and therefore for accurate ranking of observations by relative risk. However, strong ranking performance does not automatically imply well-calibrated probability estimates. The raw output scores reflect the model's ordering of exposures, but their absolute magnitude is not guaranteed to correspond to empirical default frequencies.

In a credit risk setting, this distinction is economically material. Probability of Default (PD) estimates inform provisioning, pricing, capital allocation, and risk appetite decisions. For these purposes, calibration is essential: exposures assigned similar predicted PDs should exhibit realised default frequencies consistent with those probabilities. In other words, among loans with predicted PD close to 20%, the observed default rate should be approximately 20% in expectation.

To align predicted probabilities with observed frequencies, Platt scaling [44] is applied as a post-hoc calibration procedure. This approach fits a logistic regression model that maps the raw XGBoost probability scores to calibrated probabilities

via a monotonic transformation. Formally, the calibrator learns parameters (α, β) such that

$$\hat{p}_{\text{cal}} = \sigma(\alpha + \beta s), \tag{5.1}$$

where s denotes the raw model score and $\sigma(\cdot)$ is the logistic function.

The calibration model is fitted using predictions from an intermediate classifier trained exclusively on the training partition. The predicted probabilities generated on the validation set are then used as inputs to estimate the logistic calibration mapping:

```
y_val_raw = xgb_model.predict_proba(X_val)[:, 1].reshape(-1, 1)

calibrator = LogisticRegression()
calibrator.fit(y_val_raw, y_val)
```

Listing 5.11: Platt scaling calibration fitted on the validation set

A manual logistic regression is preferred over `CalibratedClassifierCV` in order to maintain explicit control over the calibration sample and the separation between the base classifier retraining step and the score-to-probability mapping.

As discussed in Section 5.2.3, the validation set was also used to guide hyperparameter selection via Optuna. Consequently, it is not strictly independent for calibration purposes, which may introduce a mild optimistic bias in the calibration estimates. This limitation is accepted as a practical compromise given the limited size of the available dataset.

The calibration parameters are not re-estimated after the final retraining step. The logistic mapping is learned once on the intermediate classifier and then kept fixed during test evaluation and portfolio scoring. This approach limits the risk of overfitting in the calibration layer while allowing the base classifier to benefit from the full available development sample.

Calibration quality on the held-out test set is assessed through a reliability diagram and is discussed in Chapter 6.

Discrimination Performance on the Test Set

The discriminative performance of the calibrated model is evaluated on the held-out test set using the AUC–ROC and the Gini coefficient, which represents the standard normalised discrimination measure within the Basel IRB framework [8]. It is worth noting that calibration does not affect the ranking of observations and therefore leaves the AUC unchanged; the metric is nonetheless reported on the calibrated model for consistency. Under the ROC formulation, the Gini index is defined as:

$$\text{Gini} = 2 \cdot \text{AUC} - 1 \tag{5.2}$$

The quantitative results achieved on the test set are reported in Chapter 6, together with a discussion of the performance differential between the validation and test partitions and the statistical factors that contribute to it.

Threshold Analysis and Confusion Matrix

Binary classifiers require a *decision threshold* that converts continuous probability scores into discrete class predictions. This choice encodes an explicit trade-off between false negatives (missed deteriorations) and false positives (unnecessary monitoring interventions). In credit risk applications, this trade-off is asymmetric: failing to detect a deteriorating exposure typically entails higher financial and reputational costs than generating an additional monitoring signal. For this reason, lower thresholds are often preferred in early warning settings. The threshold analysis conducted on the test set — including the confusion matrix at the selected operating point — is presented in Chapter 6.

Precision-Recall Analysis

In the context of credit risk, where the dataset exhibits significant class imbalance, the ROC curve may provide an overly optimistic assessment of model performance: by incorporating true negatives into its calculation, it rewards correct classification of the majority class as heavily as detection of the minority class. To provide a more focused evaluation centred on the quality of the deterioration signal, the Precision-Recall (PR) curve is employed as a complementary diagnostic. Unlike the ROC curve, the PR curve is computed exclusively from true positives, false positives, and false negatives — making it insensitive to the size of the negative class and therefore more informative under class imbalance. The PR curve and the resulting Average Precision score are reported in Chapter 6.

5.2.5 Model Packaging and MLflow Logging

Once training and calibration are complete, the two components — the XGBoost classifier and the Platt scaling calibrator — are encapsulated in a single deployable artefact through a custom MLflow [29] `PythonModel` subclass. This design allows a single `mlflow.pyfunc.load_model()` call at inference time to return an object whose `predict()` method accepts a raw feature DataFrame and returns calibrated PD estimates directly, without requiring the calling application to manage the two-step scoring pipeline. The wrapper additionally re-applies the `category` dtype to `flag_forborne_old` before scoring, since serialisation contexts such as REST endpoints or Spark UDFs do not preserve pandas dtype metadata:

```
1 class CalibratedXGBModel(mlflow.pyfunc.PythonModel):
2     def __init__(self, xgb_clf, calibrator):
```

```
        self.xgb_clf    = xgb_clf
        self.calibrator = calibrator

    def predict(self, context, model_input):
        df = model_input.copy()
        if "flag_forborne_old" in df.columns:
            df["flag_forborne_old"] = (
                df["flag_forborne_old"].astype("category")
            )
        raw = self.xgb_clf.predict_proba(df)[:, 1].reshape(-1, 1)
        return self.calibrator.predict_proba(raw)[:, 1]

with mlflow.start_run(run_name="XGB_CreditRisk_Robust"):
    mlflow.log_params(best_params)
    mlflow.log_metric("auc_test",  auc_test)
    mlflow.log_metric("gini_test", gini_test)
    mlflow.pyfunc.log_model(
        artifact_path  = "calibrated_credit_model",
        python_model   = CalibratedXGBModel(xgb_model, calibrator),
        input_example  = X_train_full.head(5)
    )
```

Listing 5.12: Unified CalibratedXGBModel wrapper and MLflow experiment logging

Logging the model with its hyperparameters and test metrics serves two governance objectives: it ensures full reproducibility by allowing any scoring run to be traced back to the exact model version and training configuration that produced it; and it provides a centralised audit trail consistent with the model documentation requirements of regulated financial institutions.

5.2.6 Portfolio Scoring and Output Construction

Scoring is applied to the full population of active exposures. Raw XGBoost scores are passed through the Platt calibrator to produce final PD estimates, and each loan is assigned to a risk class through the traffic light function. The reference date is recorded dynamically at scoring time:

```
probs_raw_all = xgb_model.predict_proba(X)[:, 1].reshape(-1, 1)
probs_cal_all = calibrator.predict_proba(probs_raw_all)[:, 1]

data_riferimento = datetime.today().strftime("%Y-%m-%d")

risultato = pd.DataFrame({
    "numero_rapporto":  df_ml["numero_rapporto"].reset_index(drop=True),
    "data_riferimento": data_riferimento,
```

```
    "pd":                probs_cal_all,
    "target_reale":      df_ml["target"].reset_index(drop=True)
})

risultato["semaforo"] = risultato["pd"].apply(applica_semaforo)
risultato = risultato.sort_values(by="pd", ascending=False)
```

Listing 5.13: Full portfolio scoring with loan identifier and reference date

The continuous PD estimates are mapped to a discrete four-tier risk classification based on fixed probability thresholds. These thresholds were defined in alignment with the institution's internal monitoring policy and risk appetite framework, and are intended to translate model output into actionable operational categories for credit officers:

Table 5.3: Traffic light risk segmentation thresholds

Risk Class	Label	PD Threshold
Green	SANO	$\text{PD} < 10\%$
Yellow	WATCHLIST	$10\% \leq \text{PD} < 15\%$
Orange	ATTENZIONE	$15\% \leq \text{PD} < 30\%$
Red	ALTO_RISCHIO	$\text{PD} \geq 30\%$

Four design choices in the output schema deserve explicit justification. `numero_rapporto` serves as the primary key enabling downstream joins with the loan management system. `data_riferimento` makes the table append-compatible across successive monthly runs, building a longitudinal history of PD estimates without overwriting previous snapshots. `target_reale` is populated with the observed outcome label derived from the January snapshot, which is available in this experimental context because both monthly periods were processed during the study. In a prospective production deployment, this field would be null at scoring time and populated retrospectively once the following month's labels become available, allowing the table to serve directly as input to future validation procedures without structural modification. The descending sort by predicted PD ensures credit officers are immediately presented with the highest-risk exposures. The descriptive statistics of the resulting portfolio segmentation are reported in Table 6.4 in Chapter 6.

5.2.7 Persistence to the Databricks Catalog

The scored DataFrame is converted to a Spark DataFrame and written to a managed Delta Lake table within the Unity Catalog:

Table 5.4: Schema of the `credit_predictions` output table

Column	Type	Description
`numero_rapporto`	string	Unique loan identifier
`data_riferimento`	string	Scoring reference date (YYYY-MM-DD)
`pd`	float64	Calibrated probability of deterioration
`target_reale`	int64	Observed ground-truth label
`semaforo`	string	Risk class assignment

```
spark_df = spark.createDataFrame(risultato)
spark_df.write.mode("overwrite").saveAsTable(
    "catalog.ml_analysis.credit_predictions"
)
```

Listing 5.14: Storage of scoring output to the Databricks Unity Catalog

The `overwrite` mode maintains a single authoritative view of the most recent predictions. In a production setting with full historical traceability requirements, this would be replaced by `append` mode combined with a `data_riferimento` partition key, so that each monthly run is stored as an independent partition and the complete history of PD estimates remains queryable. The Delta Lake format supports this natively through ACID transaction guarantees and time-travel capabilities. The table is accessible to downstream consumers through the Unity Catalog access control layer, consistent with the least-privilege policies described in Section 5.1.

5.2.8 Governance, Monitoring, and Recalibration

The long-term operational validity of a credit risk model depends not only on its performance at initial deployment, but on the robustness of the processes in place to detect and respond to performance degradation over time. Three governance mechanisms are embedded in the deployment architecture, each anchored to the concrete artefacts produced by the pipeline.

Population stability monitoring: at each monthly scoring run, the distribution of input features in the new data is compared against the training distribution using the Population Stability Index (PSI) [45]. A PSI above 0.25 for any feature triggers a formal model review. The longitudinal structure of the output table — partitioned by `data_riferimento` — provides the natural substrate for tracking feature distributions across runs without additional infrastructure.

Performance backtesting: as ground-truth labels become available for previously scored loans, predicted PDs from the preceding month are compared against actual deterioration indicators, and the AUC and Gini are recomputed on a rolling basis. The `target_reale` field, populated retrospectively as described above, makes the output table directly queryable for this purpose. A sustained decline below a pre-defined discriminative power threshold initiates a retraining cycle.

Calibration monitoring: the mean predicted PD across the portfolio is compared against the observed deterioration rate on a quarterly basis. Systematic divergence triggers recalibration of the Platt scaling parameters, which can be performed without full retraining of the base XGBoost classifier, substantially reducing the operational cost of keeping the model aligned with current portfolio dynamics.

Collectively, these provisions transform the model from a static analytical output

into a living component of the institution's credit risk infrastructure, anchoring the governance framework described in Section 5.1 to the specific operational artefacts produced by the machine learning pipeline.

5.3 Time Series Forecasting

5.3.1 Forecasting Objectives and Use Cases

Beyond the governance and integration objectives addressed by the ETL framework, the Databricks platform offers native machine learning capabilities that can be applied directly to the curated data produced by the pipeline. One such capability is automated time series forecasting, which was explored in this study as an additional dimension of the platform evaluation. The present analysis was conducted as an exploratory feasibility study: the primary aim was to assess the end-to-end forecasting workflow offered by Databricks AutoML rather than to produce production-ready predictions, and the results should be interpreted accordingly.

The forecasting use case investigated in this context concerns the monthly estimation of financial outflows associated with guarantee activations within a credit guarantee institution. In this operational setting, when a guaranteed loan enters a default state, the guarantor institution is required to honour its commitment and disburse the guaranteed amount to the lending bank, net of any applicable counter-guarantee recoveries. Anticipating the volume of such outflows on a monthly horizon is operationally significant, as it directly informs liquidity planning, reserve provisioning, and risk exposure management.

5.3.2 Time Series Construction and Feature Engineering

Data Source and Target Variable Definition

The input data for the forecasting model was derived from the ODS tables produced by the ETL pipeline, specifically from the dataset containing the historical record of guarantee positions and their associated financial attributes. Each record represents a guarantee relationship between the institution and a beneficiary enterprise, and carries information about the guaranteed amount, the risk exposure net of counter-guarantees, the classification of the position, and relevant dates including those of origination, extinction, and deterioration.

The target variable — monthly financial outflow — was constructed by identifying positions that had transitioned into a deteriorated state within each calendar month, and by aggregating the net financial exposure of those positions as a proxy for the cash outflow that the institution would be required to disburse. The net exposure was computed by subtracting the counter-guarantee recovery component

from the gross guaranteed amount, and applying a floor of zero to prevent negative values that would arise from over-collateralised positions. The reference date used for temporal aggregation was the date of extinction or the date of the last recorded event associated with the position, whichever was available.

Deteriorated positions were identified through a combination of classification flags available in the data, including the anagrafica classification, the credit monitoring status, the non-performing indicator, and the deterioration type field. This multi-criteria identification approach was necessary because the classification of default is recorded redundantly across several fields in the source data, and no single field provides a complete and unambiguous signal in isolation.

Logarithmic Transformation of the Target

The raw monthly outflow series exhibits strong right-skewness, driven by a small number of months with exceptionally large outflow events. To stabilise the variance and make the series more amenable to standard forecasting models — which typically assume approximately stationary residuals — the target variable was transformed using the natural logarithm with a shift of one:

$$y_t = \ln(1 + \text{outflow}_t) \tag{5.3}$$

This transformation — implemented through the `log1p` function — exhibits two desirable properties in this context. First, it compresses the dynamic range of the series, mitigating the influence of extreme values while preserving all observations. Second, it maps zero-valued months — which occur when no position is extinguished in a given period — exactly to zero, thereby maintaining a meaningful and interpretable lower bound. Predicted values were subsequently back-transformed using the inverse operation $\exp(y) - 1$, restoring the estimates to the original monetary scale.

Outlier Capping

In addition to the logarithmic transformation, a percentile-based cap was applied to the raw outflow series prior to transformation. Monthly values exceeding the 97th percentile of the observed distribution were truncated to the cap threshold. This preprocessing step was introduced to prevent a small number of structurally anomalous months — likely associated with non-recurring macro-level events, including the disruptions of the COVID-19 pandemic — from distorting the trend and seasonality components learned by the forecasting models. The cap threshold was computed empirically from the training data using the `approxQuantile` function provided by the Spark DataFrame API.

Temporal Spine and Lag Features

To ensure that the time series passed to the forecasting engine was complete and regularly spaced, a full monthly calendar spine was generated covering the entire observation period and joined with the aggregated outflow data via a left join. Months with no recorded outflows were assigned a value of zero. This step is necessary because AutoML forecasting frameworks, including Databricks AutoML, require a contiguous and uniformly spaced time index.

The engineered dataset additionally included calendar features — month number, quarter, and year — and a one-month lag of the log-transformed target as a supplementary feature to capture short-range autocorrelation. The lag is constructed such that for each month t, the corresponding lag value is the transformed outflow observed at month $t - 1$, ensuring strict temporal separation between features and target and preventing any form of look-ahead bias.

The final training dataset covered the period from January 2019 to December 2025, yielding 84 monthly observations. This span includes the COVID-19 pandemic period, whose anomalous outflow dynamics were partially mitigated through the percentile-based cap described above. The series length is adequate to support seasonal pattern detection — requiring at minimum two to three full annual cycles — and to provide a meaningful validation window. However, it remains modest by the standards of deep learning models such as DeepAR, which typically benefit from substantially longer training histories; this constraint is acknowledged as a limiting factor in the comparative evaluation of the three forecasting algorithms, as discussed further in Section 5.3.4 and Chapter 6.

5.3.3 Databricks AutoML Forecasting Capabilities

Platform Architecture and Workflow

Databricks AutoML Forecasting provides a fully managed, no-code machine learning workflow accessible through both a graphical user interface and a Python API. The workflow accepts a Delta table as input and requires the user to specify a time column, a target column, a forecast frequency, and a forecast horizon. All remaining steps — including data validation, preprocessing, model selection, hyperparameter tuning, and experiment logging — are handled automatically by the platform.

Upon submission, the AutoML engine constructs a structured experimentation pipeline that evaluates multiple forecasting algorithms in parallel, each with different hyperparameter configurations. All runs are logged to an MLflow experiment, where they can be compared across standardised metrics, inspected individually, and registered to the Unity Catalog model registry for deployment. The best-performing model is identified automatically based on the primary metric selected by the user and is made immediately available for inference.

The configuration applied in this study specified a monthly forecast frequency, a horizon of twelve months, and symmetric mean absolute percentage error (SMAPE) as the primary evaluation metric. SMAPE was chosen because it is scale-independent — an important property when the target has been log-transformed. It should be noted, however, that the symmetry of SMAPE implies equal penalisation of over- and under-prediction; in a financial outflow context, underestimation is typically more consequential than overestimation from a liquidity management perspective. This trade-off is accepted here given the exploratory nature of the exercise, but a production deployment would warrant a careful reconsideration of the loss function in light of the institution's asymmetric cost structure. The three forecasting frameworks explored by the AutoML engine were Prophet, ARIMA, and DeepAR, each representing a distinct modelling paradigm as described in the following subsection.

Forecasting Algorithms

Prophet [9] is an additive decomposition model developed for business time series, particularly suited to data with strong seasonal patterns, irregular holiday effects, and structural trend changes. It decomposes the series into trend, seasonality, and holiday components, each modelled independently and additively combined. Prophet is robust to missing data and outliers, and supports the incorporation of country-specific holiday calendars as external regressors — a feature that was activated for the Italian calendar in several experimental runs.

ARIMA [46] (AutoRegressive Integrated Moving Average) is a classical statistical model that captures temporal autocorrelation through a combination of autoregressive terms, differencing for stationarity, and moving average components. Databricks AutoML implements an automated version that selects the optimal order parameters (p, d, q) through information criterion minimisation, removing the need for manual specification.

DeepAR [47] is a probabilistic recurrent neural network model based on the LSTM architecture, originally developed for multi-series forecasting scenarios. It learns a shared representation across multiple time series simultaneously, which makes it particularly effective when a large number of related series are available. In the single-series scenario of this study, DeepAR operates outside its intended regime: without cross-series learning, it reduces to a standard LSTM trained on a single sequence of 84 observations, which is a relatively limited training signal for a deep learning model. As a consequence, DeepAR's performance in this context is expected to be constrained relative to its multi-series potential, and any comparison with Prophet and ARIMA should be interpreted with this structural disadvantage in mind. DeepAR was nonetheless included in the evaluation to assess whether its probabilistic output — which naturally produces prediction intervals — offered

any practical advantage over the uncertainty quantification mechanisms of the statistical models.

Experiment Tracking and Model Registry

All experimental runs were automatically logged to an MLflow experiment within the Databricks workspace. Each run recorded the full set of hyperparameters, validation metrics, and the serialised model artifact, enabling complete reproducibility and audit traceability. The best model was registered to the Unity Catalog model registry under a versioned entry, from which it can be retrieved for batch inference or deployed to a serving endpoint for online prediction. The forecast outputs — including point predictions and prediction intervals — were written to a Delta table in the analytical schema, making them immediately queryable through Databricks SQL and integrable with the dashboard layer. The comparative results across all experimental runs are reported in Chapter 6.

5.3.4 Interpretive Limitations and Methodological Caveats

Several limitations apply to the forecasting exercise and must be clearly acknowledged before interpreting the results.

The most significant limitation concerns the construction of the target variable. The identification of cash-outflow-generating events from the available data required multiple interpretive assumptions regarding field semantics, the precedence rules among redundant classification fields, and the treatment of positions with incomplete or ambiguous deterioration records. These assumptions were made independently by the analyst without domain validation, which introduces uncertainty into the ground truth itself. In a production deployment, the outflow series would need to be derived from the institution's accounting records rather than inferred from credit classification data.

A second limitation relates to the completeness of the input data. The dataset available for this study represents a single snapshot of the portfolio, from which the temporal series was reconstructed using historical event dates embedded in each record. This reconstruction is inherently incomplete: positions that were originated and extinguished within periods not covered by the available snapshot are absent from the series, potentially causing the historical outflow figures to underestimate the true volumes in certain months.

Third, the series covers a period that includes the COVID-19 pandemic and its associated macro-economic disruptions, which introduced structural breaks and regime changes that are difficult for standard forecasting models to represent faithfully. The percentile-based cap and log-transformation applied during preprocessing mitigate but do not eliminate the influence of this period on the learned model

parameters.

Fourth, as discussed in Section 5.3.3, DeepAR operates outside its intended multi-series regime in this study. Its inclusion in the comparative evaluation should be interpreted as an assessment of the platform's integration capabilities rather than a fair benchmark of the model's predictive potential.

Finally, the exploratory nature of this exercise means that no formal backtesting procedure was applied, and the model's generalisation performance beyond the validation window has not been empirically verified. The reported SMAPE scores reflect multi-step forecast errors evaluated on a held-out portion of the training series consistent with the twelve-month horizon specified in the AutoML configuration; their representativeness of true out-of-sample accuracy over a forward-looking horizon should nonetheless be interpreted with caution.

Notwithstanding these limitations, the exercise demonstrates that the Databricks AutoML forecasting infrastructure integrates seamlessly with the data produced by the ETL pipeline, supports a diverse set of modelling paradigms within a unified experiment tracking framework, and delivers results that are immediately queryable through the analytical layer. The quantitative comparison of model performance across the three algorithms and the discussion of the most informative experimental configurations are presented in Chapter 6.

Chapter 6

Results

This chapter presents the empirical results obtained from the implementation of the ETL framework and the subsequent analytical workloads on the Databricks platform. The evaluation is organized along six complementary dimensions: execution performance and runtime behavior; pipeline scalability and reusability; operational monitoring and governance visibility; the predictive performance of the machine learning pipeline; the effectiveness of automated time series forecasting; and the end-to-end integration demonstrated through the business-oriented visualization layer. Where applicable, results are discussed in relation to the architectural choices described in the preceding chapters, with the aim of assessing their practical effectiveness in an enterprise-scale data engineering and AI context.

Figure 6.1 provides a unified view of the end-to-end architecture evaluated in this chapter, from raw file ingestion through governed ETL processing to predictive modelling and business-oriented visualization. The sections that follow discuss empirical results along the same pipeline stages depicted in the diagram.

6.1 ETL Execution and Performance Evaluation

6.1.1 Execution Time Analysis

Execution time measurements were collected across multiple runs of the complete six-task Databricks Job. Table 6.1 summarizes the observed average durations for each task.

A consistent pattern emerges across all observed runs: the most time-consuming phases are not the data transformation steps themselves, but rather the initialization stages that precede them. Most significantly, this includes the construction and validation of the execution DAG by the Delta Live Tables runtime. This fixed overhead is incurred at each pipeline invocation regardless of the volume of data

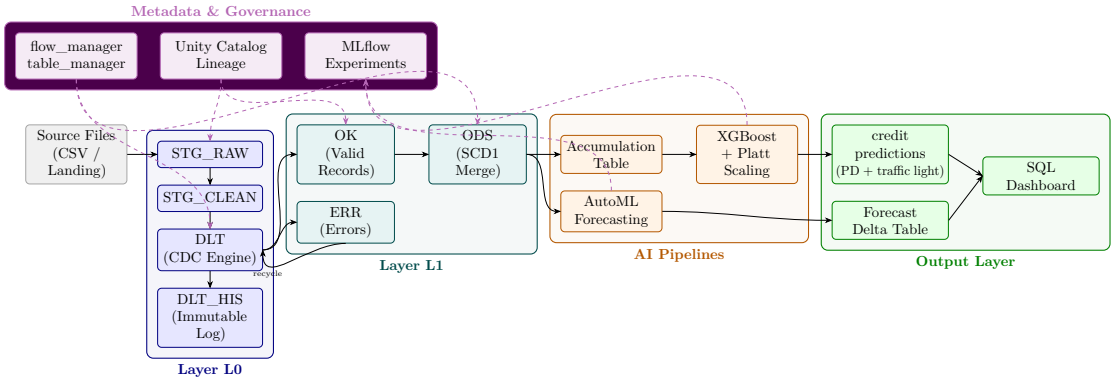

Figure 6.1: End-to-end architecture of the implemented framework. The metadata layer coordinates pipeline execution and governance across all stages. Layer L0 performs ingestion and change detection; Layer L1 applies business rule validation and ODS consolidation. The AI pipelines consume governed ODS outputs for credit risk modelling (XGBoost) and financial outflow forecasting (AutoML). The output layer exposes results through governed Delta tables and a SQL dashboard. Dashed arrows represent governance and observability flows.

Table 6.1: Observed execution times per pipeline task across multiple runs.

Task	Type	Avg. Duration (s)
audit_start	Notebook	26.0
L0 Pipeline	Delta Live Tables	287.5
audit_middle	Notebook	28.7
L1 Registry	Delta Live Tables	88.8
L1 Transactional	Delta Live Tables	178.6
audit_end	Notebook	27.0

processed, and reflects the structural cost inherent to dependency-aware declarative orchestration [28].

In relative terms, the DAG materialization phase accounts for a disproportionate share of total execution time when data volumes are small, as is the case in the experimental setup used for this study. As data volume grows in production-scale deployments, this initialization cost becomes proportionally less significant, and execution time shifts toward the transformation and merge operations in Layer L1 — particularly the `dlt.apply_changes` consolidation step and the deduplication logic in the ODS preparation stage.

The sequential dependency between the L1 Registry and L1 Transactional pipelines — enforced to preserve referential integrity as described in Section 4.5 — introduces an additional source of latency: the transactional pipeline cannot begin until the registry pipeline has fully materialized its ODS tables. In workloads characterized by large registry datasets, this constraint may become a bottleneck, and could motivate the adoption of incremental registry refresh strategies or partial sub-pipeline parallelism in future extensions.

6.1.2 Resource Allocation and Runtime Behavior

The Databricks environment was configured in serverless mode [48], which eliminates the cluster provisioning overhead associated with traditional compute configurations. Resources are allocated dynamically at pipeline startup and released immediately upon completion, making this mode particularly appropriate for intermittent workloads such as nightly ETL pipelines.

Although direct monetary cost metrics are not exposed in the Azure-managed Databricks environment used for this study, runtime duration and cluster activity serve as reliable indirect indicators of computational consumption. Per-flow output row counts extracted from the `event_log()` interface — as leveraged by the audit notebooks described in Section 4.4 — provide a quantitative basis for comparing workload intensity across pipeline runs and for detecting anomalous volume deviations that may indicate upstream data quality issues.

A notable runtime characteristic of the implemented CDC mechanism is that the cost of change detection scales with the size of the historical dataset stored in `dlt_his_<dataset>`, rather than with the volume of changes detected in any given run. As historical data grows over successive executions, the join operations underlying the set-difference comparisons may become progressively more expensive. Mitigation strategies — such as partitioning historical tables by *jobid* or applying time-bounded lookback windows — are identified as relevant optimizations for production deployments, though they fall outside the scope of this study.

6.2 Pipeline Scalability and Reusability

A central design objective of the framework is the ability to onboard new data sources without modifying the core transformation, validation, or CDC logic. The empirical results confirm that this objective is fully realized: integrating an additional dataset requires exclusively the definition of a new configuration entry in the `datasets` list for Layer L0 and a corresponding entry in `datasets_l1` for Layer L1, together with a dataset-specific business rule function. No changes to the factory functions, the CDC engine, the ODS consolidation logic, or the audit infrastructure are necessary.

This property has direct practical implications for enterprise deployments. In environments where the number of source systems grows incrementally, eliminating regression risk — that is, the possibility that onboarding a new dataset unintentionally alters the behaviour of existing pipelines — significantly reduces maintenance and testing costs.

The strict separation between configuration and implementation, enforced through the metadata-driven factory pattern described in Sections 4.3.2 and 4.3.3, ensures that each newly onboarded dataset is processed through the same validated and audited pipeline stages as all existing ones. This approach preserves behavioural consistency across executions while avoiding bespoke engineering effort for each additional source.

The DAG automatically generated by the Delta Live Tables runtime provides a further scalability indicator. As the number of registered datasets grows, the DAG expands proportionally, adding new branches of STG_RAW, STG_CLEAN, DLT, and ODS nodes for each source. The platform's dependency resolution mechanism handles this expansion transparently, preserving correct execution ordering and parallelism constraints without manual intervention.

Reusability also extends to the validation layer. The business rule pattern implemented in `business_rules.py` is inherently composable: individual rule columns can be added, removed, or modified for a specific dataset without affecting the validation logic applied to other sources. Similarly, type enforcement mappings in the `cleaning` configuration dictionary can be updated independently per dataset, accommodating heterogeneous source schemas within a unified processing model.

6.3 Monitoring and Operational Visibility

6.3.1 Audit Infrastructure: Observed Behavior

The governance infrastructure described in Section 4.4 produced consistent and coherent outputs across all pipeline runs. The `flow_manager` table correctly recorded the full lifecycle of each execution: the L0 record was opened at initialization,

closed with `STATUS = 0` and `LOAD = 1` upon L0 completion, and the L1 record was subsequently opened and closed with `LOAD = 5` at workflow conclusion. The `SRC_JOBID`–`TRG_JOBID` linkage was preserved correctly across all runs, confirming the integrity of the cross-layer lineage chain.

The `table_manager` table accumulated row-count records for each DLT flow across both pipeline layers. Tracking `NUM_ROWS` per table per run confirmed expected volume behavior: STG_RAW and STG_CLEAN tables reflected the total number of records present in the landing zone at each execution, while the CDC tables (`dlt_<dataset>`) produced only the delta detected between consecutive runs, with zero-row outputs correctly recorded when no changes were present in the source files.

The real-time DAG visualization in the Databricks UI provided immediate status visibility during pipeline execution, with node coloring correctly reflecting the pending, running, completed, and failed states across all registered tables. No unexpected failures or dependency resolution errors were observed across the executed runs.

6.3.2 Unity Catalog Lineage: Observed Coverage

Unity Catalog lineage was automatically captured for all tables registered within the governed namespace across both pipeline layers. The lineage graph correctly traced the full transformation chain — from raw landing zone files through STG_RAW, STG_CLEAN, DLT, OK, and ODS tables — without requiring any manual annotation or documentation.

In the context of the implemented framework, this lineage coverage provides two operationally significant results. First, impact analysis is immediately actionable: a schema change in any source file can be traced forward to all affected downstream tables within the Unity Catalog UI, without consulting pipeline code or external documentation. Second, the full audit trail from raw source to ODS is available for compliance purposes, satisfying data provenance requirements without additional infrastructure.

6.4 Machine Learning Performance and Model Validation

6.4.1 Discriminative Performance

The XGBoost classifier was evaluated on a held-out test set with a deterioration prevalence of 8.02%. On this set, the model achieved an AUC of 0.7092, a Gini coefficient of 0.4184, and an Average Precision (AP) of 0.409. According to Basel

Committee benchmarks [8], a Gini above 0.30 is generally considered acceptable for low-default portfolios, while values above 0.40 are regarded as good; the obtained value of 0.4184 therefore falls within the good range, consistent with well-performing short-horizon credit risk models applied to SME portfolios. The interpretation of the AP score relative to the class-prevalence baseline is discussed in detail in Section 6.4.5, in the context of the Precision-Recall curve.

The observed gap between the validation AUC (0.8392) and the test AUC (0.7092) requires careful interpretation. The test set contains only ten positive cases out of 130 total observations, yielding 1,200 possible positive–negative pairs underlying the AUC computation. Since the AUC measures the probability that a randomly chosen positive instance is ranked above a randomly chosen negative one, a single misclassified positive eliminates all 120 pairs in which it participates, shifting the AUC downward by exactly $120/1{,}200 = 0.10$ points. The observed decrease of 0.13 points is therefore consistent with approximately one to two misclassified positive cases, well within the statistical variability inherent in evaluating binary classifiers on small and imbalanced samples, rather than constituting definitive evidence of systematic overfitting. Future iterations should adopt stratified k-fold cross-validation [49] (e.g., $k = 5$) over the full dataset, which would yield a mean AUC with associated standard deviation and provide a variance-aware performance estimate robust to this small-sample instability.

6.4.2 Hyperparameter Optimization

Bayesian hyperparameter optimization via Optuna converged after 40 trials, stabilizing validation AUC at approximately 0.84 (0.8392). Table 6.2 reports the optimal configuration identified by the search.

Table 6.2: Optimal hyperparameter configuration identified by Optuna.

Hyperparameter	Optimal Value
`n_estimators`	210
`max_depth`	5
`learning_rate`	0.0627
`subsample`	0.6733
`colsample_bytree`	0.9985
`min_child_weight`	7

Gain-based feature importance derived from the final model identifies `flag_forborne_old`, `perc_utilizzo_fido`, and `flag_gia_scaduti` as the dominant predictors. This ranking is partially consistent with the mutual information analysis conducted prior to training, with one notable divergence:

`flag_forborne_old` ranks substantially higher in gain-based importance than in the marginal mutual information scores. This discrepancy is consistent with the known behaviour of gain-based metrics, which capture multivariate interaction effects and tend to attribute greater importance to variables whose splits effectively reduce residual impurity when evaluated in combination with other features. Overall, both analyses align with established credit risk theory, providing convergent evidence of genuine predictive signal in the selected variables.

6.4.3 Calibration Quality

Calibration quality is evaluated on the held-out test set using a reliability diagram (Figure 6.2), which plots the mean predicted probability against the observed default frequency across five probability bins. The choice of five bins represents a compromise between resolution and statistical reliability: a larger number of bins would produce estimated default rates based on one or zero positive observations per bin, rendering the diagram unstable and largely uninformative given the limited number of positive cases in the test set.

A perfectly calibrated model lies on the 45-degree diagonal; systematic deviations above or below this line indicate, respectively, underestimation or overestimation of risk within specific probability ranges. Bin-level observed default rates are subject to sampling variability and should be interpreted with appropriate statistical caution; the diagram therefore provides an indicative rather than definitive assessment of calibration quality, complementing the discrimination metrics reported above.

6.4.4 Threshold Analysis and Operating Point Selection

Table 6.3 reports the confusion matrix decomposition at three candidate decision thresholds on the test set, illustrating the precision–recall trade-off available to credit officers.

Table 6.3: Confusion matrix decomposition at three decision thresholds on the test set (10 positive, 120 negative cases).

Threshold	TP	FP	FN	TN	Recall	Precision
10%	6	19	4	101	60.0%	24.0%
20%	5	9	5	111	50.0%	35.7%
30%	4	5	6	115	40.0%	44.4%

At the 10% threshold, the model correctly identifies 6 out of 10 deteriorated loans (recall = 60%), at the cost of 19 false positives. This configuration maximises sensitivity but generates a substantial monitoring burden on credit officers. At the

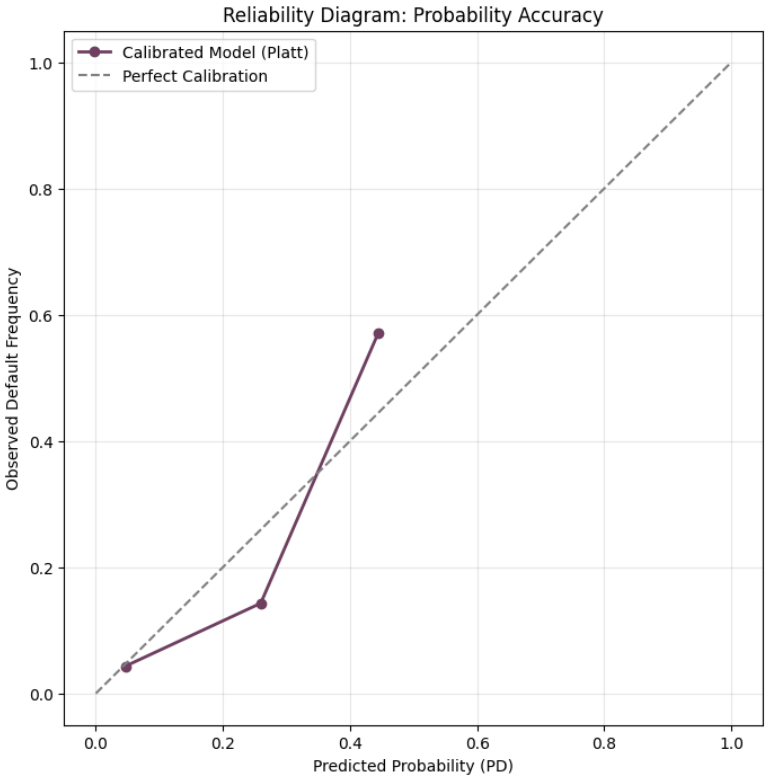

Figure 6.2: Reliability diagram for the calibrated model on the test set. Each point represents mean predicted probability against observed default rate within a bin. The dashed line represents perfect calibration.

20% threshold, false positives decrease to 9 while recall remains at 50%, achieving a meaningfully better precision (35.7% vs. 24.0%) while sacrificing only one correctly detected deterioration — a substantially more efficient operating point. At 30%, 4 deteriorations are intercepted (recall = 40%) with only 5 false positives, yielding the highest precision (44.4%) at the cost of missing 6 out of 10 deteriorated exposures.

A threshold of 20% is adopted as the reference operating point. This choice reflects the asymmetric cost structure of credit risk interventions discussed in Section 5.2.3: failing to flag a deteriorating exposure carries substantially higher financial and reputational costs than generating an unnecessary monitoring signal. The 20% threshold captures this asymmetry by prioritising recall (50%) while maintaining a precision (35.7%) well above the class prevalence baseline (8.02%), and a false positive volume (nine loans) that remains manageable under realistic monitoring constraints. The corresponding confusion matrix is shown in Figure 6.3.

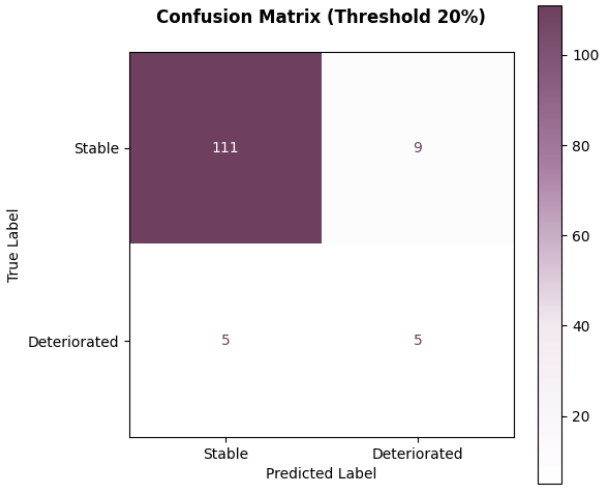

Figure 6.3: Confusion matrix at the 20% decision threshold on the test set.

At the selected threshold, the model achieves a precision of 0.36, a recall of 0.50, and an F1-score of 0.42 on the positive class. On the dominant negative class, precision reaches 0.96 with a recall of 0.93, yielding an F1-score of 0.94. The weighted average F1-score across both classes is 0.90, and overall accuracy on the test set is 89%. These results are consistent with expectations for a short-horizon early warning system applied to a low-default portfolio, where structural class imbalance and limited positive observations naturally constrain absolute precision levels.

6.4.5 Precision-Recall Analysis

Figure 6.4 reports the Precision-Recall curve for the calibrated model on the test set. The model achieves an Average Precision (AP) score of 0.409. Since a no-skill classifier would achieve an AP equal to the positive class prevalence of approximately 0.08, the model's AP represents a more than fivefold improvement over the random baseline, confirming that the classifier captures substantial discriminative signal on the minority class beyond what the ROC curve alone would indicate.

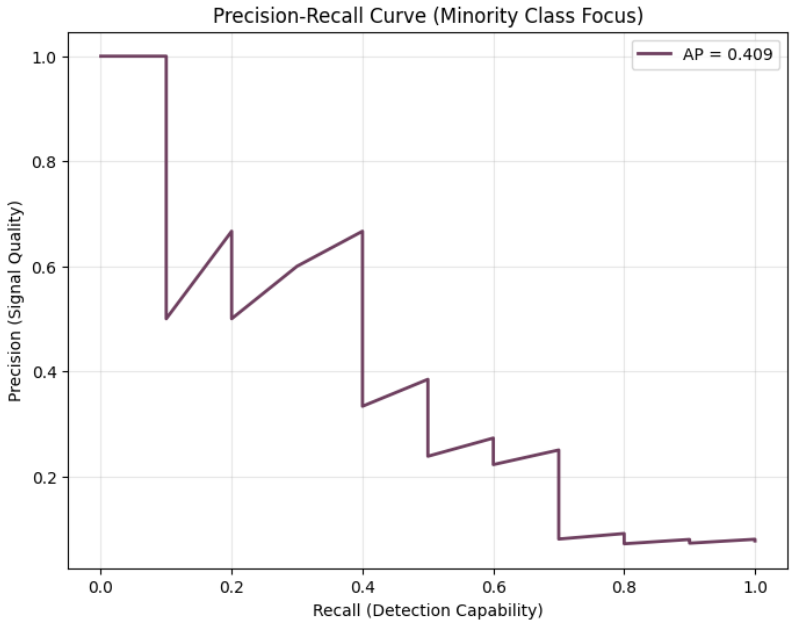

Figure 6.4: Precision-Recall curve for the calibrated model evaluated on the test set. The horizontal dashed line represents the no-skill baseline equal to the positive class prevalence (8.02%).

The characteristic sawtooth pattern reflects the discrete nature of the test set: each step in recall corresponds to a single positive case being correctly ranked, while intervening drops in precision occur when negative cases are included as the threshold is lowered. Taken together with the ROC analysis, the PR curve confirms that the model provides a meaningful early warning signal with discriminative power substantially above the no-skill baseline across the full range of decision thresholds.

6.4.6 Risk Segmentation of the Full Portfolio

Following Platt scaling, the calibrated PD estimates were used to assign each loan to one of four risk classes according to the traffic light thresholds defined in

Table 5.3. Table 6.4 reports descriptive statistics of the resulting segmentation across the full 648-loan portfolio.

Table 6.4: Descriptive statistics of calibrated PD estimates by risk class across the full 648-loan portfolio.

Risk Class	Count	Mean PD	Std	Min	Median
ALTO_RISCHIO	52	40.3%	5.0%	30.3%	40.8%
ATTENZIONE	28	22.6%	4.4%	15.5%	21.7%
WATCHLIST	26	12.3%	1.4%	10.2%	12.3%
SANO	542	3.9%	1.3%	2.9%	3.4%

The segmentation is monotonic and economically coherent. The ALTO_RISCHIO class exhibits a mean PD of 40.3% against 3.9% for SANO exposures — a tenfold difference in central tendency — with low intra-class dispersion across all segments confirming well-defined and stable risk boundaries. High-risk exposures account for approximately 8.0% of the portfolio (52 loans), closely aligned with the observed deterioration rate of 8.02% in the labelled sample. This alignment between the proportion of exposures flagged at the highest severity level and the empirical default frequency provides further qualitative validation of calibration quality, corroborating the reliability diagram discussed in Section 6.4.3.

6.5 Time Series Forecasting Results

An AutoML forecasting experiment was conducted comparing Prophet, ARIMA, and DeepAR on the monthly financial outflow series, with SMAPE as the primary evaluation metric. Table 6.5 reports the full set of validation results across all evaluated configurations. MDAPE is reported only for Prophet runs in which the metric was successfully computed by the Databricks AutoML evaluation framework: ARIMA and DeepAR do not expose this metric within the platform's standardized evaluation pipeline, and a valid MDAPE estimate was not produced for the *bright-shad-713* run.

The best-performing model across all runs was the Prophet configuration *skittish-ram-893*, which achieved a validation SMAPE of 8.70% and MDAPE of 8.41%. A SMAPE below 10% is generally regarded as indicative of good forecast accuracy for monthly financial series; the obtained value therefore suggests that the best Prophet configuration captures the dominant dynamics of the outflow series with reasonable precision, notwithstanding the dataset limitations acknowledged in Section 5.3.4.

Prophet consistently outperformed both ARIMA and DeepAR across all evaluated configurations, with ARIMA occupying an intermediate position and DeepAR

Table 6.5: Validation performance across all AutoML forecasting runs. MDAPE is reported only for Prophet configurations in which the metric was successfully produced; ARIMA and DeepAR do not expose this metric within the Databricks AutoML evaluation framework.

Run	**Algorithm**	**Val. SMAPE**	**Val. MDAPE**
bright-shad-713	Prophet	0.1063	–
welcoming-bird-197	Prophet	0.0912	0.0901
amusing-shrike-643	Prophet	0.0922	0.0920
invincible-frog-139	Prophet	0.0899	0.0882
skittish-ram-893	Prophet	0.0870	0.0841
casual-fly-313	ARIMA	0.0989	–
bemused-ape-670	DeepAR	0.1063	–
fun-finch-85	DeepAR	0.1052	–
monumental-skunk-589	DeepAR	0.1081	–
traveling-horse-140	DeepAR	0.1096	–

yielding the highest error rates. This outcome is consistent with the single-series nature of the forecasting task: as discussed in Section 5.3.4, DeepAR's cross-series learning advantage is not exploitable in this setting, which is precisely the condition under which the model tends to underperform relative to univariate methods. ARIMA's intermediate performance reflects its ability to capture short-range autocorrelation but its limited flexibility in representing the non-linear trend and seasonal components present in the series.

The best model was registered to the Unity Catalog model registry and its twelve-month point forecasts with 80% prediction intervals were written to the output Delta table, making them immediately available for consumption by the dashboard layer described in the following section.

6.6 Business-Oriented Visualization and End-to-End Integration

The final stage of the implemented pipeline integrates governed ETL outputs, calibrated PD predictions, and time series forecasts into a native Databricks SQL Dashboard. All metrics displayed are sourced exclusively from records that passed the Layer L1 business rule validations, ensuring that the data quality guarantees established upstream are transparently inherited by the client-facing outputs without requiring additional filtering logic at the dashboard level.

The dashboard is organized into three functional panels. The first panel provides a portfolio-level risk overview, displaying the distribution of loans across the four traffic light risk classes as a bar chart, the aggregate mean PD across the full portfolio as a scalar KPI, and the count of exposures crossing the 20% decision threshold adopted as the reference operating point in Section 6.4.4. The second panel presents entity-level detail: for each loan, the predicted PD, risk class assignment, and key financial indicators sourced directly from the ODS — outstanding balance, days in arrears, and credit utilisation rate — are displayed in a sortable table ordered by descending PD, together with the ground-truth label where available, allowing credit officers to immediately focus monitoring attention on the highest-risk exposures and verify the plausibility of each prediction against the underlying financial evidence. The third panel presents the time series forecasting output, displaying the historical monthly outflow series alongside the twelve-month Prophet forecast and its 80% confidence interval, enabling risk managers to contextualise current portfolio conditions within the projected liquidity outflow trajectory.

Exposing both the predicted PD and the underlying financial indicators within the same interface enables domain experts to assess the plausibility of each prediction in context and exercise informed judgment in their risk management activities. Analytical responsibility thus remains with the human decision-maker, while the

platform provides the governed data infrastructure that makes this evaluation tractable at scale.

This end-to-end integration — from raw ingestion through governed transformation, predictive modelling, and business visualization — within a single unified workspace constitutes the primary empirical validation of the lakehouse value proposition [6] discussed in Section 2.2.3. The elimination of cross-system synchronization reduces governance complexity and operational overhead, while full traceability is preserved across the entire data lifecycle through the Unity Catalog lineage and audit infrastructure described in Sections 4.4 and 6.3.2.

Chapter 7

Discussion

This chapter critically evaluates the design decisions, empirical outcomes, and inherent limitations of the proposed ETL framework. Rather than summarizing results already presented in Chapter 6, the discussion focuses on interpreting those findings in a broader context: assessing the generalization potential of the architectural model, articulating the theoretical and practical grounds for the observed strengths, situating the framework relative to traditional ETL approaches, and identifying the constraints that bound its current applicability. The goal is to provide a rigorous and honest appraisal of the contribution, informing both its immediate practical use and its future development.

7.1 Architectural Generalization and Reusability

A central claim of the proposed framework is its domain-agnostic nature — a qualification that applies specifically to structured and semi-structured tabular data processed in batch or micro-batch mode, as discussed further at the end of this section. The layered architecture, metadata-driven orchestration, and execution-driven change detection mechanism are not inherently tied to risk analytics or financial data. Rather, they constitute a general-purpose design pattern applicable to any enterprise data integration context in which governance, traceability, and incremental processing are primary requirements. This claim rests on the strict separation maintained between the structural pipeline logic and the dataset-specific configuration. The factory functions that register Delta Live Tables pipelines, the CDC engine that computes set differences between consecutive execution states, the error recycling mechanism, and the SCD1 merge logic in the ODS are all implemented as reusable, parameterized components that operate uniformly across all datasets regardless of their schema, volume, or semantic domain. Domain specificity is confined exclusively to two artifacts: the column-type mapping in the `cleaning`

configuration dictionary and the business rule functions in `business_rules.py`. Both are isolated from the core pipeline infrastructure and can be defined independently for each new source without any coupling to existing datasets. This architectural boundary has implications that extend beyond mere code reuse. In enterprise environments where data governance must be applied consistently across heterogeneous business domains — finance, operations, human resources, supply chain — a unified structural template ensures that governance properties such as lineage tracking, error auditability, and execution reproducibility are not selectively applied but are enforced by construction for every dataset onboarded through the framework. The alternative, which is the proliferation of individually crafted pipelines for each domain, systematically undermines governance consistency and increases the long-term cost of compliance. The layered L0–L1 architecture also provides a portable integration model for organizations transitioning from legacy data warehouse environments to modern lakehouse platforms. The staging and cleansing logic in L0, the validation and consolidation logic in L1, and the clear interface between them — mediated by the DSO framework delta tables (`DLT_<dataset>`) and the shared *jobid* — map naturally onto the ELT paradigm characteristic of lakehouse architectures, as discussed in Section 2.2.3. Organizations that have historically relied on ETL processes pushing pre-transformed data into a data warehouse can adopt the L0–L1 pattern incrementally, beginning with the ingestion and cleansing stages and extending to semantic integration as organizational readiness allows. This incremental adoptability reduces the disruption associated with platform migration and lowers the architectural risk of large-scale re-engineering efforts. One important qualification applies to the generalization argument: the framework as implemented is optimized for structured and semi-structured tabular data arriving in batch or micro-batch mode. Extension to fully unstructured sources — such as free-text documents, audio, or sensor streams with sub-second latency requirements — would require non-trivial adaptations to the ingestion, cleansing, and CDC layers, and may necessitate different processing paradigms not currently supported by the Delta Live Tables declarative model. This boundary should be acknowledged when evaluating the framework's applicability in data-intensive domains such as IoT or natural language processing pipelines.

7.2 Strengths of the Proposed Framework

7.2.1 Deterministic and Execution-Driven Change Detection

The CDC mechanism described in Section 4.3.2 derives its operational semantics entirely from discrete pipeline execution identifiers rather than from timestamp

comparisons or database-level change tracking logs. This design choice confers a property that is often underappreciated in ETL engineering: *determinism*. Given the same sequence of *jobid* values and the same historical dataset, the CDC engine will always produce identical output, irrespective of the time at which the pipeline is executed or replayed. This guarantee is not achievable with timestamp-based CDC implementations [5], which are sensitive to clock drift, out-of-order arrival, and the ambiguity of `NULL` or missing timestamp fields in source systems.

The practical consequence of this determinism is that pipeline re-execution for recovery, debugging, or retrospective analysis does not produce spurious change records or corrupt the historical audit trail. This property is particularly valuable in regulated environments — such as financial risk management — where the ability to reconstruct the exact state of the data at any point in time may be subject to regulatory scrutiny [23, 8]. The compensating record strategy, which encodes state transitions as immutable append-only records rather than in-place updates, further reinforces this guarantee by ensuring that no prior history is overwritten during reprocessing.

7.2.2 Governance by Construction

A distinctive characteristic of the framework is that governance properties — traceability, auditability, data quality enforcement, and lineage — are embedded structurally within the pipeline architecture rather than implemented as optional post-hoc additions. The `JOBID` and `INS_TIME` audit fields are initialized at L0 ingestion and propagated unchanged through every downstream layer; the `flow_manager` and `table_manager` tables are populated as integral steps of the workflow, not as optional logging modules; and the error recycling mechanism ensures that rejected records are never silently discarded but are preserved and reprocessed transparently.

This approach contrasts with governance models in which data quality dashboards, lineage documentation, and audit logs are maintained as separate systems layered on top of the data pipeline. The separation between pipeline execution and governance recording introduces synchronization risks: governance artifacts may fall out of date if the pipeline is modified, re-executed out of sequence, or extended without corresponding updates to the documentation layer. By embedding governance within the execution fabric itself, the proposed framework eliminates this risk class entirely.

7.2.3 Separation of Concerns and Maintainability

The strict functional decomposition across pipeline layers — L0 for ingestion and change tracking, L1 for validation and integration — and across modules within each layer ensures that each component has a single, well-defined responsibility. This

adherence to the Separation of Concerns principle [35] has direct maintainability implications: a change in the cleansing logic for a specific dataset affects only the relevant entry in the `cleaning` configuration dictionary and the corresponding `STG_CLEAN` factory call, with no risk of propagating side effects to the CDC engine, the validation layer, or the ODS consolidation step. Similarly, a modification to the business rules for one dataset has no bearing on the validation logic applied to other datasets, since each rule function is independently defined and invoked.

This modularity also facilitates testing: individual pipeline components can be unit-tested in isolation by providing mock DataFrames and metadata inputs, without requiring the full pipeline infrastructure to be instantiated. In enterprise environments where continuous integration and automated regression testing are standard practices, this testability is a prerequisite for safe and reliable pipeline evolution.

7.2.4 Automated Dependency Resolution

The delegation of execution ordering to the Delta Live Tables runtime — through automatic DAG construction based on declared `dlt.read()` and `spark.readStream` dependencies — eliminates a significant class of operational errors that affect manually sequenced pipelines. In traditionally orchestrated ETL systems, the execution order of pipeline steps is often encoded implicitly in scheduling configurations or procedural scripts, making it fragile in the face of pipeline restructuring and difficult to validate without full end-to-end execution. The declarative dependency model adopted in this framework makes the execution graph explicit, machine-verifiable, and visually inspectable through the Databricks UI, reducing the cognitive load required to reason about pipeline correctness and simplifying onboarding for new team members.

7.3 Limitations of the Implementation

7.3.1 Manual File Ingestion and Landing Zone Management

As noted in Section 4.1, the experimental setup relies on manually uploaded CSV files in the landing zone volume, rather than automated ingestion from live source systems. While this approach was appropriate for the controlled conditions of a research implementation, it introduces a gap between the demonstrated capabilities of the framework and the operational requirements of a production deployment. In production, source systems would be expected to deliver data autonomously — through scheduled exports, event-driven triggers, or streaming connectors — and the ingestion layer would need to handle variable file arrival times, partial uploads, and malformed files without human intervention. Although Auto Loader

provides the technical infrastructure for automated and fault-tolerant ingestion, the operational workflows for monitoring file delivery, handling missing or late-arriving sources, and managing upstream SLA compliance were not addressed in this study.

7.3.2 Infrastructure Cost Opacity

In the Azure-managed Databricks environment used for this implementation, granular cost metrics at the pipeline or task level were not directly accessible through the standard workspace monitoring interfaces. This constraint reflects the configuration of the specific workspace used in this study, which operated under a shared academic subscription in which billing export and cost attribution features were not enabled; production-grade deployments with dedicated subscriptions would have access to granular cost data through Azure Cost Management APIs or the Databricks billing export functionality.

As a consequence of this limitation, it was not possible to quantify computational costs at a fine-grained level — for example, per pipeline run, per dataset, or per logical processing stage. This constrains the ability to conduct detailed cost–benefit analyses that would be required for production capacity planning, such as comparing serverless and cluster-based execution under different data volume scenarios or estimating the marginal cost associated with onboarding an additional dataset. Addressing this gap represents a relevant extension for future deployments operating under full subscription access.

7.3.3 Single-Platform Dependency

The implementation makes deliberate use of Databricks-specific capabilities — including Delta Live Tables, the `dlt.apply_changes` API, Unity Catalog governance primitives, and the `event_log()` function — that are not available in a platform-agnostic form. While this maximizes the utilization of native platform features and reduces implementation complexity, it also introduces a vendor dependency that limits portability. Migrating the framework to an alternative cloud-native platform — such as a workflow orchestrator like Apache Airflow[1] combined with a transformation layer such as dbt[2] — would require re-implementing the Delta Live Tables pipeline definitions, the CDC output architecture, and the audit infrastructure using different abstractions. The core algorithmic logic — set-difference-based CDC, SCD1 merge, error recycling — is platform-agnostic and could be preserved, but the operational integration binding it to the Databricks execution model would require non-trivial re-engineering.

[1] `https://airflow.apache.org`

[2] `https://www.getdbt.com`

7.3.4 Incomplete Implementation of Layer L2 and MDM

The implementation scope was deliberately bounded at Layer L1 ODS consolidation, leaving the Layer L2 publication layer and the Master Data Management enrichment stage as architectural projections rather than realized components. As a consequence, the end-to-end data lifecycle — from raw ingestion to fully dimensional, business-aligned analytical structures — is not demonstrable within the current implementation. The OUT layer, which translates enriched MDM data into L2-aligned structures, and the star schema fact and dimension tables that would constitute the final analytical layer, exist in the framework specification described in Chapter 3 but were not instantiated within the scope of this thesis. This limits the empirical evaluation of the framework's performance and governance properties at the analytical publication stage, and means that the business-oriented dashboards described in Section 6.6 draw directly from ODS tables rather than from fully modeled dimensional structures. The architectural choices made at L0 and L1 — including surrogate key governance and the OUT staging structure — are nonetheless designed to anticipate and facilitate this future extension, as discussed in Chapter 3.

7.3.5 Machine Learning Pipeline Integration

The machine learning outputs referenced in the decision support dashboard were produced through a pipeline described in Chapter 5 that, while fully functional and governed through MLflow experiment tracking, was not formally integrated with the ETL pipeline infrastructure at the execution level. This reflects a deliberate scoping decision: the primary objective of this thesis was to demonstrate the feasibility of the ETL framework on Databricks and to validate its compatibility with downstream analytical workloads, rather than to establish a production-grade MLOps pipeline with end-to-end automated execution. Specifically, the provenance chain between the curated data produced by the ETL framework and the model outputs displayed in the dashboard — traversing feature engineering, model training, calibration, and inference — is not currently traceable through a single unified lineage graph within Unity Catalog.

It should be noted that the accumulation table design described in Section 5.1 was introduced precisely to bridge the semantic gap between the ODS layer and the machine learning consumption layer, ensuring that validated records are exposed to the data science team through a governed, append-only feature store. However, this bridge addresses data availability and access control but does not yet constitute full end-to-end provenance traceability from raw source to model output within a unified governance graph.

In a production context, establishing this traceability is a prerequisite for regulatory compliance and model governance: any model output consumed in a

risk management decision should be fully traceable back to the raw source data from which it was derived, with a documented and auditable record of every transformation applied along the way. Closing this gap through formal MLflow-to-Unity-Catalog integration and the definition of a feature store aligned with the ODS layer represents a significant avenue for future development, and would complete the governance chain currently spanning from raw ingestion to ODS consolidation.

7.4 Comparison with Traditional ETL Approaches

Traditional batch-oriented ETL systems — predominantly implemented through procedural scripting, proprietary ETL tools, or stored procedure chains — differ from the proposed framework along several structural dimensions that are worth examining explicitly.

The most fundamental distinction concerns the handling of data state transitions. Legacy ETL pipelines typically apply update-in-place logic: when a record changes in the source system, the corresponding row in the target table is overwritten, and the prior state is lost unless a separate historization mechanism is explicitly implemented. The proposed framework, by contrast, encodes all state transitions as immutable compensating records that are appended to the historical delta tables without modifying existing data. This append-only model ensures that any prior state of any entity can be reconstructed at any point in time by filtering the history table on the relevant *jobid* range, without reliance on backup systems or snapshot tables maintained outside the pipeline.

A second distinction concerns the management of execution dependencies. In traditional ETL architectures, the sequencing of pipeline steps is typically enforced through scheduler configurations, procedural control flows, or explicit wait conditions between jobs. This externalized dependency management is brittle: it requires manual synchronization with the pipeline structure and can silently produce incorrect results if the execution order drifts from the intended sequence due to scheduling delays or partial failures. The declarative dependency model adopted in this framework — where execution ordering is derived automatically from data lineage at pipeline construction time — eliminates this fragility by making dependencies intrinsic to the pipeline definition rather than external to it.

Third, traditional ETL pipelines frequently embed transformation logic, business rules, and dataset-specific parameters directly within procedural code, creating a tight coupling between configuration and implementation that increases the cost of change and limits reusability. The metadata-driven configuration approach adopted in the proposed framework enforces a clean separation between these concerns: transformation logic is implemented once as a parameterized, dataset-agnostic

function, while dataset-specific parameters are externalized into structured configuration dictionaries. This inversion of control is a prerequisite for the scalability and maintainability properties demonstrated in Section 6.2.

Fourth, error handling in traditional ETL pipelines is often implemented as a binary gate: records that fail validation are either rejected entirely — frequently without preservation — or allowed to pass with a warning flag that may or may not be acted upon. The error recycling mechanism described in Section 4.3.3 represents a more nuanced approach: rejected records are preserved in a governed error table and reintroduced into subsequent pipeline runs, where they are revalidated against current data conditions. This design acknowledges the reality of enterprise data environments, where validation failures often arise not from permanent data defects but from transient conditions such as the late arrival of referenced master data. By treating error resolution as an iterative process rather than a terminal event, the framework reduces data loss and improves the completeness of the analytical outputs over successive pipeline runs.

Finally, it is important to acknowledge a dimension in which traditional ETL approaches retain a practical advantage: operational accessibility. The framework proposed in this thesis requires proficiency in PySpark, the Delta Live Tables programming model, and Unity Catalog governance primitives — a skill set that is not universally available in enterprise data teams. Established ETL tools such as Informatica, Talend, or SQL Server Integration Services offer lower-code interfaces and more accessible learning curves, which may be decisive factors in organizations with limited Spark expertise or constrained engineering capacity. The architectural advantages of the proposed framework are therefore most valuable in contexts where Databricks competency is already present or being actively developed, and where the governance and scalability requirements justify the higher initial investment in platform expertise.

Taken together, these distinctions suggest that the proposed framework represents not merely an incremental improvement over traditional ETL approaches, but a structural shift in the design philosophy underlying enterprise data integration — from imperative, state-mutating, monolithically configured pipelines toward declarative, immutable, governance-oriented architectures that treat data quality and auditability as first-class engineering requirements rather than post-hoc additions. The extent to which this shift translates into measurable improvements across the full L0–L2 lifecycle remains an open empirical question, contingent on the resolution of the limitations identified in Section 7.3 — most urgently the completion of the L2 publication layer, the automation of the ingestion infrastructure, and the formal integration of the machine learning pipeline within the unified governance graph. These directions are addressed in the concluding chapter.

Chapter 8

Conclusions and Future Developments

This concluding chapter synthesizes the principal contributions of the thesis, reflects on the research objectives established at the outset, and identifies the most promising directions for future development. Rather than restating the technical details presented in preceding chapters, the discussion is oriented toward the significance of the work as a whole and the trajectory along which the framework could evolve to meet the demands of fully production-grade enterprise deployments.

8.1 Summary of Contributions

This thesis presented the design, implementation, and evaluation of a governed, multi-layer ETL framework built on the Databricks lakehouse platform. The work was motivated by the practical challenges of enterprise data integration in environments characterized by source heterogeneity, strict governance requirements, and the need to support both operational reporting and advanced analytical workloads from a unified data foundation.

The primary objective was to study the Databricks platform in sufficient depth to evaluate its architectural components and data engineering capabilities within the context of a structured, industrialized integration framework. This objective was met through the concrete realization of Layer L0 and Layer L1, encompassing the full pipeline from raw file ingestion through to ODS consolidation, as described in Chapter 4, and through a critical assessment of the platform's governance and observability primitives as applied to a real-world data integration scenario.

The specific contributions of the work are the following.

The **layered L0–L1 integration architecture** establishes a reusable structural template for governance-oriented data platforms. By enforcing a strict separation

between ingestion and change tracking (L0) and semantic validation and consolidation (L1), the framework ensures that each processing stage has well-defined inputs, outputs, and responsibilities, enabling independent evolution, testing, and maintenance of each layer without systemic risk.

The **execution-driven CDC model** represents a methodologically rigorous approach to incremental change detection. By grounding change semantics in pipeline execution identifiers rather than timestamp heuristics or database-level logs, the mechanism achieves deterministic reproducibility and full compatibility with the append-only, immutable storage model characteristic of lakehouse architectures. The compensating record strategy ensures that the complete history of entity state transitions is preserved and queryable without reliance on external snapshot mechanisms.

The **metadata-driven orchestration strategy** demonstrates that configuration–implementation separation is not merely a software engineering best practice but a practical prerequisite for scalable data platform governance. The ability to onboard new datasets through configuration changes alone — without modifying or retesting core pipeline logic — directly reduces the operational cost and governance risk associated with platform growth in enterprise environments, as confirmed by the empirical results reported in Section 6.2.

The **embedded audit and governance infrastructure**, implemented through the `flow_manager` and `table_manager` tables and the three-notebook audit lifecycle, establishes a persistent, queryable observability layer that is structurally integrated with pipeline execution rather than maintained as a separate system. This design ensures that governance records are always consistent with actual pipeline behavior, regardless of whether the pipeline is executed on schedule, replayed for recovery, or extended with new datasets.

The **analytical access governance layer**, implemented through the dedicated `ml_analysis` schema, the accumulation table strategy, and the least-privilege permission structure described in Section 5.1, demonstrates that the transition from a governed ETL platform to an analytical consumption layer can be accomplished without introducing uncontrolled data exposure. This design enforces data minimisation by construction and establishes a governed, auditable interface between the operational pipeline and the data science team, ensuring that only analytically relevant attributes are surfaced while personal identifiers and operationally sensitive fields remain protected.

The **machine learning pipeline** developed atop the governed ODS outputs demonstrates that the framework effectively supports complex predictive workloads beyond technical data integration. The XGBoost credit deterioration model achieves a Gini coefficient of 0.4184 and an Average Precision of 0.409 on the held-out test set — results discussed in detail in Section 6.4 — representing a fivefold improvement in discriminative power over the random baseline and placing the

model within the range regarded as good under Basel Committee benchmarks for low-default portfolios. By calibrating these outputs via Platt scaling, the model produces probability estimates directly interpretable as empirical default frequencies, providing actionable risk signals for credit officers.

The **time series forecasting exercise** confirms that the governed data produced by the ETL framework maintains the quality and structure required for production-grade predictive analytics beyond the credit risk domain. The Prophet-based financial outflow forecasting pipeline, evaluated against ARIMA and DeepAR within the Databricks AutoML infrastructure, achieved a validation SMAPE of 8.70% — below the 10% threshold generally regarded as indicative of good forecast accuracy for monthly financial series — while demonstrating seamless integration with the Delta table outputs of the ETL framework and the Unity Catalog model registry.

The direct linkage between validated ODS data and the risk monitoring dashboard, described in Section 6.6, ensures that the data quality guarantees enforced upstream are transparently inherited by the metrics consumed by domain experts, closing the gap between raw data ingestion and business-oriented insights.

Collectively, these contributions establish a reproducible, governance-aware data lifecycle that addresses the core requirements of enterprise data integration — traceability, quality, extensibility, and analytical readiness — within a modern cloud-native environment. The empirical evidence gathered across the execution, scalability, monitoring, and analytical dimensions evaluated in Chapter 6 confirms that the Databricks platform is sufficiently mature and governance-capable to support a structured, industrialized ETL framework meeting enterprise-grade requirements, within the scope boundaries identified in Chapter 7.

8.2 Future Developments

The framework as implemented constitutes a robust and well-governed foundation, but several architectural extensions and operational enhancements would be necessary to transition it from a research prototype to a fully production-grade enterprise data platform. Among these, the completion of the L2 layer and the MLOps integration represent the extensions with the deepest architectural implications for the framework's design; the remaining developments — ingestion automation, orchestration enhancement, cost monitoring, and streaming support — are primarily operational in nature and can be pursued independently of one another.

8.2.1 Ingestion Automation and Source System Integration

The most immediate gap between the current implementation and a production deployment is the reliance on manual file uploads in the landing zone, as discussed in Section 7.3.1. Replacing this with automated ingestion would require the definition of source system connectors — leveraging cloud object storage event notifications, message queue integrations, or enterprise data integration platforms such as Azure Data Factory [50] — that deliver files to the landing zone autonomously according to agreed delivery schedules or event triggers. In addition to the technical connectors, production ingestion requires operational procedures for monitoring file arrival, alerting on missing or late-arriving sources, and handling malformed or duplicate deliveries without human intervention. Auto Loader already provides the incremental processing semantics needed to support these scenarios; the remaining work is primarily operational and integration-focused rather than architectural.

8.2.2 Completion of Layer L2 and MDM Enrichment

The Layer L2 publication layer and the Master Data Management enrichment stage are the most structurally significant extensions to the current implementation. Realizing Layer L2 would involve the implementation of the OUT transformation logic — translating enriched MDM data into structures aligned with the target dimensional schema — and the definition of the star schema fact and dimension tables that constitute the final analytical layer, as specified in the framework architecture described in Chapter 3. This extension would close the end-to-end data lifecycle and enable empirical evaluation of the framework's performance and governance properties at the analytical publication stage, including query performance on dimensional schemas and the behavior of the ODS-to-L2 loading pipeline under production-scale data volumes. The MDM enrichment layer, which introduces surrogate key governance and contextual attribute augmentation, is a prerequisite for L2 and would additionally improve the referential consistency and join performance of the dimensional structures.

8.2.3 Advanced Orchestration and Cross-System Coordination

The current orchestration model, implemented through a single Databricks Job, is well-suited to self-contained pipelines operating within a single workspace. In enterprise environments where the ETL framework must coordinate with external systems — upstream source databases, downstream consuming applications, model serving infrastructure, or data products owned by other teams — a more capable orchestration layer would be beneficial. External orchestration tools such as Apache

Airflow[1] or Azure Data Factory could complement the Databricks Jobs model by providing cross-system workflow management, SLA monitoring, conditional branching based on upstream delivery confirmation, and integration with enterprise scheduling and alerting infrastructure. This would decouple the ETL pipeline execution from the availability of specific Databricks resources and enable the framework to participate in broader organizational data workflows without requiring all components to be hosted within the same platform.

8.2.4 MLOps Integration and Feature Store Alignment

As discussed in Section 7.3.5, the machine learning outputs referenced in the analytical dashboard are not currently connected to the ETL pipeline through a formally governed, end-to-end lineage chain within Unity Catalog. Closing this gap requires two complementary developments.

First, formal integration between MLflow experiment tracking and the Unity Catalog lineage graph would make every model output traceable back to its source data, completing the provenance chain from raw ingestion to inference without relying on separate, unconnected provenance stores. Second, the definition of a governed feature store aligned with the ODS and MDM layers would standardise the feature engineering process across model iterations, ensuring that training and inference consume identical, versioned representations of the underlying data assets. Model training, experiment tracking through MLflow, and deployment to a model serving endpoint could then be integrated as downstream tasks within the Databricks Job, creating a single governed pipeline spanning data ingestion, transformation, feature engineering, model training, and inference. This end-to-end integration would satisfy the provenance and reproducibility requirements increasingly mandated by regulatory frameworks governing the use of machine learning in financial risk management [51].

8.2.5 Cost Monitoring and Resource Optimization

Addressing the infrastructure cost opacity identified in Section 7.3.2 would require integration with Azure Cost Management APIs or the Databricks billing export infrastructure to associate monetary cost estimates with individual pipeline runs and dataset processing stages. This capability would enable data-driven capacity planning, cost allocation across business domains, and optimization of the trade-off between serverless and cluster-based execution under different volume profiles. In particular, the cost behavior of the CDC historical join operations — which scale

[1] `https://airflow.apache.org`

with the size of the accumulated delta history tables, as noted in Section 6.1.2 — warrants systematic measurement to determine whether time-bounded lookback windows or partition-based pruning strategies offer meaningful cost reductions without compromising change detection completeness.

8.2.6 Extension to Streaming and Near-Real-Time Workloads

The current framework is designed for batch and micro-batch ingestion patterns, where data arrives in discrete files at scheduled intervals. In use cases where lower latency is required — such as real-time fraud detection, intraday risk monitoring, or event-driven data products — the ingestion and CDC layers would need to be extended to support continuous streaming semantics. Databricks Structured Streaming and the Delta Live Tables streaming table model already provide the technical primitives for this extension; the primary challenge lies in adapting the execution-driven CDC logic, which currently relies on discrete *jobid* boundaries, to a model where changes must be detected and propagated continuously without natural execution boundaries. Approaches based on watermarking [52], stateful stream processing, and micro-batch accumulation windows are candidate solutions that would preserve the deterministic and auditable properties of the current CDC design while supporting near-real-time operational requirements.

The realization of these extensions would progressively close the gap between the framework as demonstrated and a fully operational enterprise data platform. The architectural foundations laid in this thesis — the layered processing model, the metadata-driven orchestration, the governance-by-construction principle, and the append-only audit trail — are designed to accommodate these developments without requiring fundamental redesign. More broadly, the work presented here offers an affirmative and empirically grounded answer to the central research questions posed in Section 1.2: Databricks, when engaged through a structured, governance-oriented architectural framework rather than as a collection of ad-hoc tools, demonstrates the maturity, flexibility, and observability required to support enterprise-grade data integration and analytical workloads within a unified, cloud-native environment.

Bibliography

[1] D. Laney. *3D Data Management: Controlling Data Volume, Velocity and Variety*. META Group Research Note. Feb. 2001 (cit. on pp. 1, 5).

[2] M. Armbrust et al. «A View of Cloud Computing». In: *Communications of the ACM* 53.4 (Apr. 2010) (cit. on pp. 1, 10, 14).

[3] M. Zaharia et al. «Apache Spark: A Unified Engine for Big Data Processing». In: *Communications of the ACM* 59.11 (Nov. 2016) (cit. on pp. 1, 3, 14, 17).

[4] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. «Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores». In: *Proceedings of the VLDB Endowment* 13.12 (2020) (cit. on pp. 1–3, 11, 14–16).

[5] R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. Third. Indianapolis, IN: Wiley, 2013 (cit. on pp. 2, 6–9, 22, 25, 26, 31, 44, 52, 109).

[6] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia. «Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics». In: *Proc. 11th CIDR Conference*. Chaminade, CA, Jan. 2021 (cit. on pp. 2, 3, 6, 11, 12, 14–20, 37, 106).

[7] D. J. Hand and W. E. Henley. «Statistical Classification Methods in Consumer Credit Scoring: A Review». In: *Journal of the Royal Statistical Society: Series A* 160.3 (1997) (cit. on pp. 3, 4, 16, 17).

[8] Basel Committee on Banking Supervision. *International Convergence of Capital Measurement and Capital Standards: A Revised Framework*. Tech. rep. Basel, Switzerland: Bank for International Settlements, June 2006 (cit. on pp. 3, 20, 81, 98, 109).

[9] S. J. Taylor and B. Letham. «Forecasting at Scale». In: *The American Statistician* 72.1 (2018) (cit. on pp. 3, 4, 16, 90).

[10] Databricks, Inc. *What is Auto Loader?* Accessed: 2026-03-03. 2025. URL: `https://docs.databricks.com/aws/en/ingestion/cloud-object-storage/auto-loader/` (cit. on pp. 3, 34).

[11] Databricks, Inc. *Lakeflow Jobs*. Accessed: 2026-03-03. 2026. URL: https://docs.databricks.com/aws/en/jobs/ (cit. on pp. 3, 61).

[12] M. Golfarelli, S. Rizzi, and I. Cella. «OLTP and OLAP Data Integration: A Review of Feasible Implementation Methods and Architectures for Real Time Data Warehousing». In: *Proc. IEEE SoutheastCon 2005*. Ft. Lauderdale, FL, Apr. 2005 (cit. on p. 5).

[13] Z. G. Alfughi and A. Ouda. «The Future of Data Warehousing: Trends, Technologies, and Challenges in the Era of Big Data, Cloud Computing, and Artificial Intelligence». In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 10.5 (2024) (cit. on p. 5).

[14] S. Azzabi. «Data Lakes: A Survey of Concepts and Architectures». In: *Computers* 13.7 (2024) (cit. on pp. 5, 10).

[15] J. Schneider, C. Gröger, A. Lutsch, H. Schwarz, and B. Mitschang. «The Lakehouse: State of the Art on Concepts and Technologies». In: *SN Computer Science* 5 (2024) (cit. on pp. 5, 11, 14).

[16] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. San Francisco, CA: Morgan Kaufmann, 1992 (cit. on pp. 6, 15).

[17] E. F. Codd, S. B. Codd, and C. T. Salley. *Providing OLAP to User-Analysts: An IT Mandate*. Tech. rep. Codd and Associates, 1993 (cit. on p. 6).

[18] W. H. Inmon. *Building the Data Warehouse*. Fourth. Indianapolis, IN: Wiley, 2005 (cit. on pp. 7, 11, 20).

[19] J. Dixon. *Pentaho, Hadoop, and Data Lakes*. Blog post, Pentaho, https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/. Accessed: 2026-03-03. Oct. 2010 (cit. on pp. 10, 11).

[20] V. Mayer-Schönberger and K. Cukier. *Big Data: A Revolution That Will Transform How We Live, Work, and Think*. New York, NY: Houghton Mifflin Harcourt, 2013 (cit. on p. 10).

[21] S. Heilgeist, S. Aier, and R. Winter. «Making Sense of the Data Lake: A Systematic Literature Review». In: *Enterprise Modelling and Information Systems Architectures* 16 (2021) (cit. on pp. 10, 11, 15).

[22] DAMA International. *DAMA-DMBOK: Data Management Body of Knowledge*. Second. Basking Ridge, NJ: Technics Publications, 2017 (cit. on pp. 12, 13, 20–22, 24, 27).

[23] European Parliament and Council of the European Union. *Regulation (EU) 2016/679 on the Protection of Natural Persons with Regard to the Processing of Personal Data (GDPR)*. Official Journal of the European Union, L 119/1, https://eur-lex.europa.eu/eli/reg/2016/679/oj. Apr. 2016 (cit. on pp. 12, 13, 15, 16, 66, 109).

[24] State of California. *California Consumer Privacy Act of 2018 (AB-375)*. California Legislative Information, https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375. Accessed: 2026-03-03. 2018 (cit. on p. 12).

[25] U.S. Department of Health and Human Services. *Health Insurance Portability and Accountability Act of 1996 (HIPAA), Public Law 104-191*. U.S. Government Publishing Office, https://www.hhs.gov/hipaa. Accessed: 2026-03-03. 1996 (cit. on pp. 12, 13).

[26] S. S. Das. «Transforming Data: Role of Data Catalog in Effective Data Management». In: *International Journal for Research Trends and Innovation* 9.12 (2024) (cit. on p. 12).

[27] P. Buneman, S. Khanna, and W. C. Tan. «Why and Where: A Characterization of Data Provenance». In: *Proc. 8th International Conference on Database Theory (ICDT)*. London, UK, Jan. 2001 (cit. on pp. 13, 21, 27).

[28] Databricks, Inc. *Databricks Documentation*. Accessed: 2026-03-03. 2026. URL: https://docs.databricks.com (cit. on pp. 13, 14, 17, 19, 35, 95).

[29] M. Zaharia et al. «Accelerating the Machine Learning Lifecycle with MLflow». In: *IEEE Data Engineering Bulletin* 41.4 (2018) (cit. on pp. 15, 17, 82).

[30] Databricks, Inc. *Unity Catalog Documentation*. Accessed: 2026-03-03. 2026. URL: https://docs.databricks.com/aws/en/data-governance/unity-catalog/ (cit. on pp. 16, 33).

[31] Delta Lake Contributors. *Delta Sharing: An Open Protocol for Secure Data Sharing*. Accessed: 2026-03-03. 2021. URL: https://github.com/delta-io/delta-sharing (cit. on pp. 16, 67).

[32] B. Dageville et al. «The Snowflake Elastic Data Warehouse». In: *Proc. ACM SIGMOD International Conference on Management of Data*. San Francisco, CA, June 2016 (cit. on p. 17).

[33] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. «Dremel: Interactive Analysis of Web-Scale Datasets». In: *Proceedings of the VLDB Endowment* 3.1 (2010) (cit. on p. 17).

[34] Microsoft Corporation. *Microsoft Fabric Documentation*. Accessed: 2026-03-03. 2024. URL: https://learn.microsoft.com/en-us/fabric/ (cit. on p. 18).

[35] E. W. Dijkstra. «On the Role of Scientific Thought». In: *Selected Writings on Computing: A Personal Perspective*. Original essay dated 1974 (EWD 447). New York, NY: Springer, 1982 (cit. on pp. 21, 110).

[36] L. Libkin and L. Peterfreund. *Handling SQL Nulls with Two-Valued Logic*. arXiv preprint arXiv:2304.01987. Accessed: 2026-03-03. 2023 (cit. on p. 44).

[37] James A. Ohlson. «Financial Ratios and the Probabilistic Prediction of Bankruptcy». In: *Journal of Accounting Research* 18.1 (1980) (cit. on p. 68).

[38] S. Kaufman, S. Rosset, C. Perlich, and O. Stitelman. «Leakage in Data Mining: Formulation, Detection, and Avoidance». In: *ACM Transactions on Knowledge Discovery from Data* 6.4 (2012) (cit. on p. 70).

[39] T. Chen and C. Guestrin. «XGBoost: A Scalable Tree Boosting System». In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016 (cit. on pp. 72, 76).

[40] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. 2nd. Hoboken, NJ: Wiley-Interscience, 2006 (cit. on p. 74).

[41] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. «SMOTE: Synthetic Minority Over-sampling Technique». In: *Journal of Artificial Intelligence Research* 16 (2002) (cit. on p. 77).

[42] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. «Optuna: A Next-Generation Hyperparameter Optimization Framework». In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019 (cit. on p. 77).

[43] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. «Algorithms for Hyper-Parameter Optimization». In: *Advances in Neural Information Processing Systems*. Vol. 24. 2011 (cit. on p. 77).

[44] J. Platt. «Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods». In: *Advances in Large Margin Classifiers*. Ed. by A. J. Smola, P. Bartlett, B. Schölkopf, and D. Schuurmans. Cambridge, MA: MIT Press, 1999 (cit. on p. 80).

[45] N. Siddiqi. *Credit Risk Scorecards: Developing and Implementing Intelligent Credit Scoring*. Hoboken, NJ: Wiley, 2006 (cit. on p. 86).

[46] G. E. P. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung. *Time Series Analysis: Forecasting and Control*. 5th. Hoboken, NJ: Wiley, 2015 (cit. on p. 90).

[47] D. Salinas, V. Flunkert, J. Gasthaus, and T. Januschowski. «DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks». In: *International Journal of Forecasting* 36.3 (2020) (cit. on p. 90).

[48] Databricks, Inc. *Serverless Compute*. Accessed: 2026-03-03. 2026. URL: https://docs.databricks.com/aws/en/compute/serverless/ (cit. on p. 95).

[49] R. Kohavi. «A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection». In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. 1995 (cit. on p. 98).

[50] Microsoft Corporation. *Azure Data Factory*. https://azure.microsoft.com/en-us/products/data-factory. Accessed: 2026-03-03. 2015 (cit. on p. 118).

[51] European Banking Authority. *Report on Machine Learning for IRB Models*. Tech. rep. Available at: https://www.eba.europa.eu/sites/default/files/document_library/Publications/Reports/2021/1023168/EBA%20report%20on%20ML%20for%20IRB%20models.pdf. Accessed: 2026-03-03. European Banking Authority, 2021 (cit. on p. 119).

[52] T. Akidau et al. «The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing». In: *Proceedings of the VLDB Endowment* 8.12 (2015) (cit. on p. 120).