Data Lakehouse Platforms for Machine Learning: A Comparative Evaluation

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Software Engineering
June 2026
Waltteri Mynttinen

UNIVERSITY OF TURKU
Department of Computing

Waltteri Mynttinen: Data Lakehouse Platforms for Machine Learning: A Comparative Evaluation

Master of Science (Tech) Thesis, 57 p., 1 app. p.
Software Engineering
June 2026

The data lakehouse is a modern data management architecture that combines the reliability of data warehouses with the flexibility of data lakes. It is promoted as a strong foundation for machine learning workloads, but the architecture itself can be implemented in various ways across different platforms. This thesis examines the data lakehouse architecture and its support for machine learning, and compares how different platforms approach its implementation.

This thesis combines a literature review with empirical research. Five lakehouse features most relevant to the machine learning lifecycle were identified by reviewing relevant literature: ACID transactions, time travel, schema enforcement, feature stores, and pipeline orchestration. An evaluation framework was defined to assess the features along two axes: implementation mechanism and observable behavior. The framework was applied in a case study comparing an identical machine learning pipeline implemented on an open source stack, Databricks, and Snowflake.

The case study showed that all three platforms provided the same essential data lakehouse functionality while utilizing different approaches for implementing the features. The most notable difference was related to the division of responsibility between the platform and the engineer. The managed platforms absorbed configuration and lifecycle decisions, whereas the open source implementation left them to the engineer. The choice of platform therefore depends less on which features are available than on how each platform implements them.

Keywords: data lakehouse, machine learning, MLOps, Databricks, Snowflake, Apache Iceberg, cloud computing

Contents

1 Introduction

Machine learning has become one of the most influential and important technologies of recent times in both industry and research. Its growth is driven by the increasing availability of large datasets and the advancement of cloud computing. Increased machine learning workloads place demands on data infrastructure, which cause undesired effects and issues with traditional data management platforms. A conventional approach to solving these problems has been a two-tier architecture, where data lakes filled with large quantities of raw unstructured data are fed into a curated data warehouse. This arrangement produces its own issues, most notably data staleness, latency, and management overhead.

The data lakehouse architecture has emerged to address these problems by combining the reliability features of data warehouses, such as ACID transactions and data versioning, with the flexibility and cheap storage options provided by data lakes. The lakehouse is considered a separate architecture instead of a product provided by a managed platform. It is possible to construct a lakehouse architecture from independent open source components as well as bought as a managed platform. While different platforms claim to implement the same lakehouse architecture and features, platform-specific quirks and various approaches for implementing the functionality vary significantly. All of these differences are not apparent from documentation alone, and can instead affect the machine learning practitioner directly when working with a data lakehouse architecture.

1.1 Research questions

This thesis analyzes data lakehouse platforms and their features for supporting machine learning workloads compared to alternative data management platforms. It identifies the lakehouse features most relevant to the machine learning lifecycle, defines an evaluation framework for them, and applies that framework in a case study comparing three identical machine learning pipelines implemented on three separate platforms. These objectives are summarized in the following three research questions (RQs):

- **RQ1: Which data lakehouse features are most relevant for supporting machine learning?**
- **RQ2: How can data lakehouse features be evaluated to compare their implementation across platforms?**
- **RQ3: How do data lakehouse platforms differ in their approach to implementing the data lakehouse architecture?**

RQ1 aims to analyze the data lakehouse architecture, and to identify central features that it offers to support machine learning workloads. RQ2 is used to define mechanisms and behaviors to evaluate the features offered by the lakehouse architecture across different implementations. RQ3 examines different data lakehouse platforms to assess their overall approach to implementing a simple machine learning pipeline to highlight differences across a defined machine learning lifecycle.

1.2 Research methods

The research questions in this thesis are answered by a combination of literature review and empirical research. Essential features of the data lakehouse architecture

are selected and an evaluation framework is created by examining relevant literature. The empirical research in this thesis is conducted with a case study consisting of a comparative evaluation of three different approaches to implement a data lakehouse architecture.

Essential literature is reviewed to identify which lakehouse features are most relevant for supporting the machine learning lifecycle. The selected features are then mapped to relevant MLOps principles which reflect their impact on machine learning most appropriately. For each selected feature, two evaluation axes are defined by reviewing platform documentation and relevant literature to form an evaluation framework to highlight differences in approach to implementing essential lakehouse functionality. The first axis investigates the implementation mechanism of the feature, while the second one describes the observable behavior of the feature.

The comparative platform comparison is conducted to gain insight into different approaches to implementing a data lakehouse and a working machine learning pipeline within it. Three separate platforms will be used to create an identical machine learning pipeline to observe platform behavior under identical conditions. The dataset and machine learning task chosen for the pipeline are simplistic by nature on purpose to keep the focus on the platform behavior and differences instead of modeling complexity and performance results.

References used in this thesis were found from several different platforms consisting of UTU Volter, IEEE Xplore, Google Scholar, and ACM Digital Library. The search words used for finding references were “data lakehouse”, “lakehouse architecture”, “machine learning”, "data warehouse", “data lake”, “challenges”, and “feature”. The selection of references was based on their relevance to the topic of this thesis and their publication date. Literature published before 2015 was ignored. In addition to academic literature, official documentation published by each platform and their tools were also consulted to offer specific technical details in the empirical part of

the thesis.

1.3 Structure

Chapter 2 of the thesis serves as a background chapter. The first part of the chapter consists of a brief history and description of relevant data management architectures. The second chapter describes the popularity of machine learning and its shift to the cloud, while also going over machine learning compatibility with data lakehouses and a description of machine learning operations.

The third chapter is used to describe how machine learning is enabled in data lakehouse environments. The first part of the chapter defines a machine learning lifecycle to offer a broader view of what its different stages include. The second part of the chapter focuses on determining essential features related to machine learning in the data lakehouse architecture. The third part of the chapter is used to define mechanisms and observable behaviors of the selected features to assess the impact of the features on the machine learning. An evaluation framework is defined in this section of the chapter, which is used to assess different lakehouse platforms later in the thesis.

The fourth chapter consists of a case study, where three different data lakehouse platforms are compared with each other to analyze their approach to implementing essential lakehouse features and how they support machine learning. The chapter's first part contains a setup description, where the dataset, machine learning model, and platforms used in the comparison are discussed. Then, each platform implementation is conducted in sequence with the established machine learning lifecycle phases to describe the details of each implementation. The final part of the chapter is reserved for the comparison, where the evaluation framework and its contents are utilized to highlight the differences between the three implementations.

Chapter 5 of the thesis discusses the results of the comparison conducted in

Chapter 4. The results are then used to answer RQ3. The chapter also includes an overall evaluation of the relevancy of the features selected and the evaluation framework to answer RQ1 and RQ2. Suggestions for future research and possible limitations are discussed at the end of Chapter 5.

Chapter 6 serves as the conclusion of the thesis. It goes over the thesis in order to summarize the findings, results, and what was done during the thesis from start to finish.

2 Background

This chapter serves as a conceptual background for the thesis. Different data management architectures and their history are introduced to judge the evolution of needs and features related to data management. This chapter also covers a brief description of machine learning, the reasons for its recent popularity, and its synergy with cloud computing. Machine learning in modern data lakehouse environments is also described in this chapter to judge their overall compatibility compared to traditional data platforms.

2.1 Data management architectures

Modern data management revolves around storing and handling large quantities of structured and unstructured data. Both types of data have created a need for unique data management architectures to serve their special characteristics and qualities. From traditional data warehouses filled with polished and structured data, to vast data lakes consisting of raw and semi-structured data, enterprise data architectures contain a lot of variance and complexity causing division regarding the two approaches[1]. The growing rift between the functionalities of data warehouses and data lakes and the lack of support for advanced analytics tasks such as machine learning have caused the emergence of a new platform, called the lakehouse, which leverages a combination of features and characteristics of the aforementioned platforms to introduce a new way of handling large quantities of data[2].

2.1.1 Data warehouse

Data warehouses are data repositories designed to combine multiple sources of data into a single location to leverage into actionable insights derived from research and reporting[3][4]. First introduced in the late 1980s and early 1990s, data warehouses were created to answer several problems in traditional business intelligence solutions related to overall usability, data quality, limited functionality, and slow performance[3]. Data warehouses have evolved a lot over the years due to the popularity of cloud-based solutions and the rapid growth of data itself[5].

A data warehouse traditionally consists of a relational database with sources consisting of online transaction processing and enterprise resource planning[3]. Data warehouses are optimized to support various analytical tasks, such as querying and reporting, to derive useful information in decision-making within organizations[4]. Extract, Transform, and Load (ETL) methods are a central part of data warehousing and how the data itself is stored[3]. ETL methods ensure a higher level of data quality by making sure the data stored in the warehouse is clean, consistent, and up to date by adhering to ACID standards: Atomicity, Consistency, Isolation, and Durability are fundamental criteria to ensure the reliability and accuracy of database transactions by maintaining data integrity while data is being read and written by multiple users or processes[3][6]. Atomicity refers to transactions being treated as singular units, where the transaction completely succeeds or no changes are made at all. This prevents data from becoming corrupted or lost due to system failures and errors. Consistency guarantees that a transaction does not create any unintended consequences but instead moves the system state from one valid state to another. Isolation ensures that concurrent transactions do not interfere with each other before they are finished. Durability secures that once a transaction is successfully executed, the changes are permanent and secure[7]. The workflow of a traditional data warehouse architecture is depicted in Figure 2.1.

Due to the evolution of advanced analytics, rising data amounts, and the emergence of cloud-based solutions, the data warehouse architecture has also evolved. The need for real-time insights and overall scalability has caused a shift from traditional on-premises data warehouses to cloud-based warehouses with robust storage and computing capabilities. Modern cloud-based data warehouse solutions provide enhanced scalability, flexibility, cost efficiency and integration with other useful cloud services while also improving data security and accessibility[5]. While evolving in some ways, data warehouses still cannot answer the ever-growing need for advanced analytics of unstructured data, which has led to the rise of alternative data platforms[1].

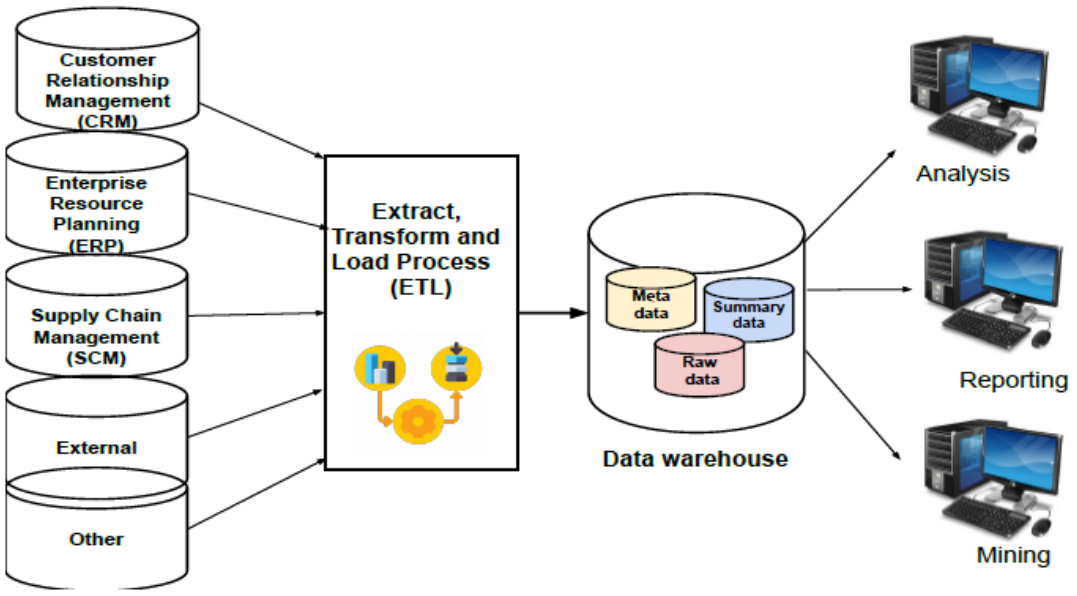

Figure 2.1: An illustrative image of a data warehouse architecture[8]

2.1.2 Data lake

Unprecedented volumes of diverse data consisting of sources such as web-based business transactions, streaming, and social media have created an avenue for generating

actionable insights from semi- and unstructured data[9]. With the amount of data increasing rapidly from multiple sources, traditional ETL processes have become less efficient over time[8][9]. The structure of modern datasets has also changed a lot. For example, unstructured datasets such as video, audio, and text documents require a different approach to storing and analyzing raw data. This has led to the rise of data lakes, which offer a low-cost solution to some of the issues caused by the overwhelming amount of data[2].

In stark contrast to data warehousing, the data in data lakes is stored in its original raw format often in large quantities. The data can vary in size and structure, as data lakes can store anything from structured data to semi-structured and unstructured data all in a centralized manner[8]. Data lakes are also beneficial for data science tasks due to the ability to utilize the storage for input and output of data analysis and machine learning tasks without the need of adhering to schema-related requirements[10]. However, traditional data lakes are non-ACID compliant, which makes routine tasks such as updates and deletes complex and inconvenient to execute reliably[6]. A data lake architecture and its functionalities are visualized in Figure 2.2.

As data lakes and data warehouses target different types of analytical tasks, this means that in many cases both must be utilized together in order to gather actionable insights from the data. A 2-tier architecture consisting of a data lake filled with source data and a pipeline delivering it into a data warehouse is often the most popular choice for an integrated solution of the two. A combined solution, while beneficial in some ways, causes issues such as increased complexity of the system's architecture and makes it also more error-prone and vulnerable to data staleness via delays[7].

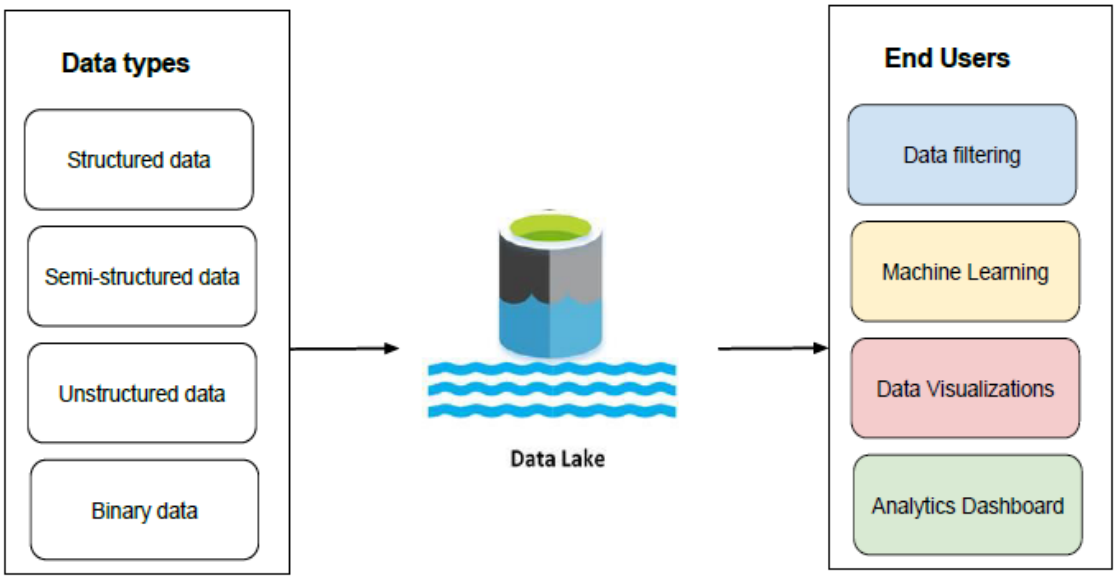

Figure 2.2: An illustrative image of a data lake architecture[8]

2.1.3 Data lakehouse

As briefly mentioned earlier, the integration of data warehouses and data lakes causes undesirable effects on system complexity, machine learning capabilities, and the quality of data. When neither platform can replace the other and the integration of said platforms causes several issues, a new platform arises: the lakehouse. Data lakehouses are described to provide traditional features of data warehouses such as ACID transactions and data versioning and combining them with open format storage capabilities to introduce a hybrid approach to data management. The benefits of a lakehouse system include improved support for machine learning, reduced complexity, and refined SQL-performance[2][1].

The term "lakehouse" was first popularized by Databricks, who have championed the data management system as a replacement for existing data architectures. This claim is justified by describing the architecture with open table technologies such as Delta Lake and Apache Iceberg for adding transactional metadata layers on top of

centralized object stores in the cloud to allow for diverse management capabilities. Open table formats are essential for the lakehouse architecture because of their ability to manage both structured and unstructured data, while ensuring high levels of data quality and reliability. Essentially, they implement database-like functionality directly on top of low-cost cloud object storage by adding a transactional metadata layer over the stored data[11][12]. By utilizing Apache Parquet as the standard file format in popular object stores in the cloud, popular machine learning libraries can easily read and query the data through the established metadata layer[2].

To further assess the lakehouse architecture, it can be divided into three distinct layers consisting of a storage layer, metadata layer, and a compute layer. This is illustrated in Figure 2.3. The storage layer consists of a data lake providing a low-cost object store option such as AWS's S3 or Azure's Data Lake Storage. The metadata layer then adds management features that allow the usage of ACID transactions, versioning, and caching with a tool such as Delta Lake or Apache Iceberg[2]. The compute layer utilizes a processing component, such as Apache Spark, to conduct tasks including reading, processing, and writing of data[7].

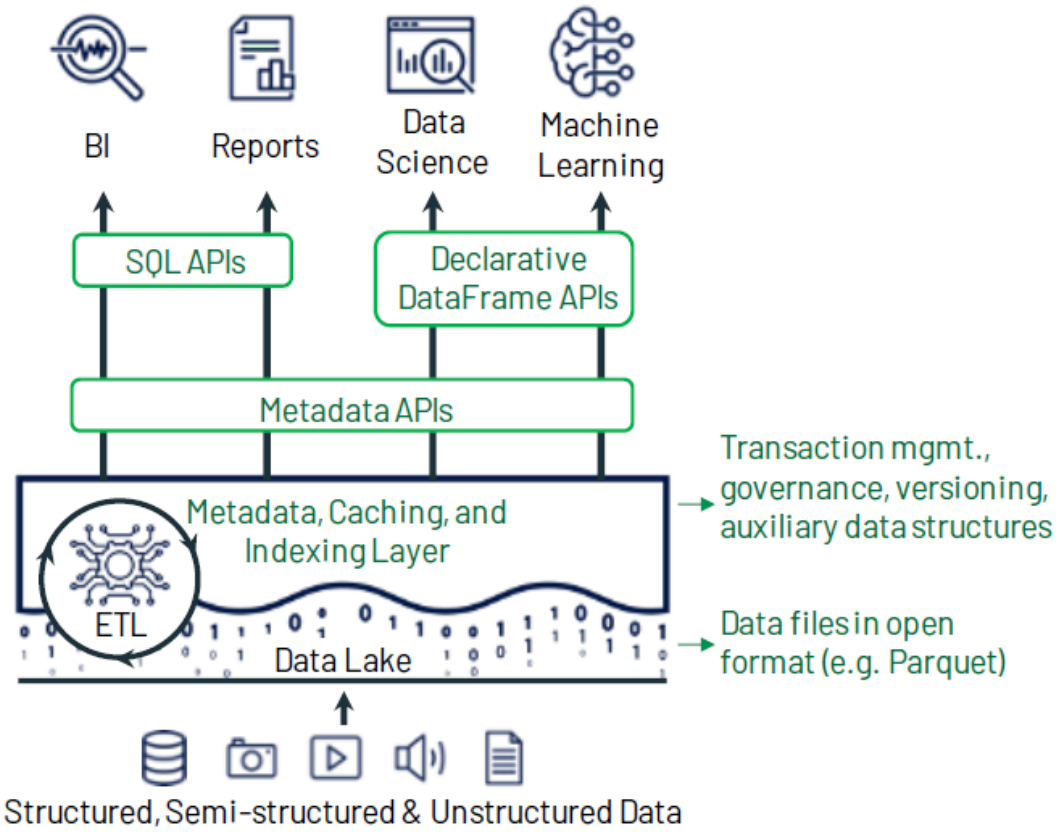

Figure 2.3: An illustrative image of a data lakehouse architecture[2]

2.2 Machine learning

Machine learning can be described as one of the most important technologies of our present time due to its capability to utilize human-like cognitive tasks that affect decision-making in all fields and markets ranging from industry to academia[13][14]. Essentially, machine learning is about building programs that adapt to new data to improve their behavior and results by generalizing and detecting patterns. As a concept, machine learning has been around for decades, but its recent popularity can be linked to the rising availability of large quantities of data, the popularity of cloud computing and the evolution of newer ML algorithms[13].

2.2.1 Machine learning in the cloud

In addition to increased data quantities and advanced ML algorithms, cloud computing has also become more popular in research and industry fields by providing provision for computing, storage, and offering flexibility[15]. The popularity of cloud computing has proven to be synergistic with machine learning workloads by offering a significant amount of benefits native to the cloud in addition to several tools and options to enhance the machine learning process[16]. This has led to a revolution in organizational capabilities to leverage data for actionable insights in order to gain a competitive advantage[16].

Cloud-native benefits related to machine learning include overall scalability, flexibility, cost efficiency, and accessibility. Popular cloud service providers also enable users to utilize machine learning capabilities without requiring advanced skills or deep knowledge related to machine learning by offering various frameworks and tools[16][13]. Machine learning has also proved to be an effective tool for fighting cloud security threats in a way that traditional methods could not. An example of this is intrusion detection, where machine learning is used to monitor user behavior and network activity to detect harmful activity and suspicious patterns[17].

Utilizing machine learning in the cloud also comes with some challenges. A common issue with implementing cloud computing services is data privacy and security. Adhering to regulations has become increasingly challenging to organizations with large quantities of data in complex cloud systems on shared infrastructure[16][18]. Another issue is related to machine learning performance, where inherent qualities of cloud environments can negatively affect performance through network latency and resource contention[18].

2.2.2 Machine learning in data lakehouse environments

Two-tier architectures consisting of data lakes filled with raw data and data warehouses filled with curated data are often hindered by a myriad of issues: data staleness, latency, tool incompatibility, and management overhead[2][1]. Data staleness and latency are caused by delays in ETL-processes, where data is first moved into a lake and then again into a warehouse, causing the possibility of data scientists working with out-of-date information[2]. Many popular machine learning frameworks such as TensorFlow and PyTorch do not work well with traditional data warehouses due to their requirement of high-throughput access to big datasets via complex non-SQL code[2]. Warehouses store data in proprietary storage formats that are not directly accessible by external tools causing unnecessary data extraction overhead[1][7].

Data management tasks such as loading and preprocessing often cause notable management overhead, where a majority of runtime is eaten up by these tasks instead of actual machine learning[1]. Another major issue is training-serving skew, where separate copies of data are maintained in data lakes and warehouses, which create inconsistencies. Training-serving skew causes the features used to train a machine learning model to differ from the data available during production inference, which causes a significant amount of production machine learning failures[1].

The data lakehouse architecture seeks to improve machine learning processes by reducing the aforementioned issues by running warehouse functionality directly on top of open file formats, such as Apache Parquet[2][1]. Parquet is a columnar format, which means that query engines can optimize the feature extraction workflow by reducing data transfer volumes compared to row-oriented formats[1]. Open file formats also allow the bypassing of slow SQL interfaces by streaming data directly from low-cost object storages[7]. Lakehouse architectures also implement ACID transactions through the transactional metadata layer on top of the open file format

in order to eliminate undesirable effects such as training-serving skew and data staleness[7][2].

Data lakehouse metadata layers also provide data versioning and time travel by tracking monotonically increasing version numbers for table modifications, which is essential for reproducibility and retraining[11][1]. Other benefits include unified batch and stream processing, where data collections are supported both as sources and destinations. This allows for incremental processing pipelines that are able to ingest new data in almost real time for live inference while also supporting heavy batch training runs on the same infrastructure[7].

2.2.3 Machine learning operations

To combat various machine learning challenges, principles from software paradigms such as DevOps began making their way into machine learning lifecycles. The most popular one, MLOps, is an engineering paradigm focusing on end-to-end conceptualization, implementation, deployment, monitoring, and scalability of machine learning solutions[19]. Its primary goal is to automate and enhance the machine learning lifecycle to reduce human intervention in repetitive tasks while also bridging the gap between development and operations[19][20]. The integration of MLOps with cloud-native technologies optimizes resource allocation and ensures system resilience. Data lakehouse architectures are highly compatible with MLOps principles and they inherently support them through core features such as unified data management and pipeline automation[21].

MLOps provides a wide range of technical and business benefits ranging from operational efficiency, improved model reliability, and enhanced collaboration. The benefits related to operational efficiency include rapid deployment through automated pipelines, and reduced manual effort by automating tasks such as data preparation and testing. The use of CI/CD tactics allow for frequent and reliable software

and model releases[19][20]. The ability to reproduce machine learning experiments and obtain consistent results is another core principle of MLOps due to its importance for debugging and validation[19][22]. Continuous monitoring and testing ensure the model's compliance with desired accuracy levels before and after deployment[20][19]. MLOps also supports various governance, security, and compliance requirements by enabling traceability and auditing through versioning of data, models, and code[19][23]. Another perk of MLOps is enhanced collaboration, which is a result of bridging domain silos between data scientists, software developers, and other IT professionals by embracing a collaborative culture[19].

Core MLOps principles in the context of data lakehouses and this thesis are data reliability and quality, reproducibility, auditability, testing, collaboration, automation, and observability. ML pipelines are concurrent by nature, because of the overlap of executions of ingestion jobs, training runs, and feature materializations. Data reliability ensures that those operations produce consistent results regardless of execution order or possible overlap[19][24]. Incoming data should match the expected schema, and writes should be isolated and atomic. This affects data quality directly, which is a central MLOps principle that is supported by lakehouses[25]. Reproducibility is achieved through the ability to reconstruct certain scenarios and training runs that have occurred during the workload. This is achieved through time travel capabilities of lakehouse environments, which is a core principle of MLOps[19]. Versioning and time travel also enforces auditability by making past decisions and operations available as needed, which is essential for governance and compliance requirements[19]. Another perk of MLOps is enhanced collaboration, which is a result of bridging domain silos between data scientists, software developers, and other IT professionals by embracing a collaborative culture and enabling simultaneous working on machine learning tasks[19]. Pipeline orchestration tools are common in lakehouse environments, which support automation and remove the need for a lot of

manual steps in the machine learning workflow[26]. These tools also offer visibility on each step of the workflow to support observability and monitoring[19].

3 Enabling machine learning in data lakehouse Environments

In this chapter, a machine learning lifecycle is defined to understand the different stages of machine learning workloads. Then, essential data lakehouse features are selected, described, and mapped across the established phases of the machine learning lifecycle to form a feature framework for understanding how each feature affects the machine learning process. An evaluation framework is also defined in this chapter to select analyzable attributes for the chosen lakehouse features to highlight the differences in implementation and approach of the platforms compared later on in this thesis.

3.1 Machine learning lifecycle

To assess machine learning enablement, it is important to understand the phases of the machine learning lifecycle. Traditional software development lifecycle structures are often unsuited to describe machine learning development due to its versatile nature and unique characteristics. Machine learning tasks are data-centric, experimental, and they require continuous adaptation to changing elements[27][28]. This has inspired the creation of multiple machine learning lifecycle structures to adequately address the needs and qualities of machine learning workloads. Based on a synthesis of established machine learning lifecycle models, the lifecycle used in

this thesis is defined in the following phases: data understanding, data preparation, modeling, evaluation, deployment, and monitoring[27][28][29]. The division of lifecycle stages can be seen in Figure 3.1.

The data understanding phase determines the project's scope and examines whether the available data can be used to support it[29]. Common tasks in this phase include defining success criteria, performing a feasibility study, identifying and collecting possible relevant data sources, and verifying data quality through visualization[29][28]. Understanding the data and acknowledging the feasibility of the project is a crucial step for successful machine learning and the following data preparation phase.

The next stage consists of the cleaning, labeling, and validation of the data. The goal is to produce a clean, finalized dataset for modeling, which often also makes it the most time-consuming part of the lifecycle[29]. Inaccurate, noisy, or inconsistent records are removed to ensure quality training input[27]. For supervised learning tasks, the data is also labeled to assign context to raw data[28]. Another important part of the data preparation phase is feature engineering, where the most informative features are selected and extracted to improve model performance[29]. Other tasks in this phase include data augmentation and standardization to ensure the consistency of the data[29].

The modeling phase is where the actual machine learning happens. A machine learning algorithm is selected and trained by iteratively adjusting parameters to minimize error and optimize performance[27][28]. Reproducibility is prioritized by documenting the iterative aspects of the process such as the version of the dataset, random seeds, and hyperparameters[29].

After selecting and training the model, the next step is to thoroughly test the model with unseen data[27][28][29]. The model and its qualities are then evaluated against the initial success criteria to assess its overall functionality[29]. The

qualities examined include metrics such as accuracy, robustness, and interpretability. Robustness measures the model's resiliency to noise or adversarial inputs, while interpretability is used to assess the model's decision-making process[29].

In the deployment stage, the validated model is integrated into a production environment or targeted device to make real-time predictions[29]. One important thing to note is that the production data possibly does not resemble the training data, which can lead to undesired effects on model accuracy[29]. Possible automation pipelines related to machine learning operations and the reducing of manual overhead also happen during this stage[28].

Once a machine learning model has been deployed, it also must be continuously monitored in order to detect performance degradation[28][27]. This can happen because of data distribution shifts, hardware aging, and system updates[29]. Monitoring tasks include the tracking of input signals and performance metrics to detect when the model requires updating or retraining[29]. Continuous training is implemented through triggers, where quality drops invoke feedback loops that automatically retrain the model on newer data[19].

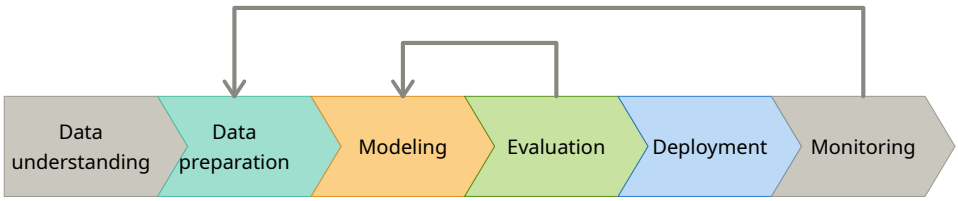

Figure 3.1: Visualization of the machine learning lifecycle[28]

3.2 Key features

As briefly discussed earlier, the data lakehouse architecture aims to optimize the machine learning process by offering various features to eliminate problems caused by complex two-tier architectures consisting of traditional data warehouses and data lakes. Essential lakehouse features can be conveniently listed and described by their relevancy to the machine learning lifecycle. Each feature can also be linked to core MLOps principles, which were defined in the background chapter. A mapping of the lakehouse features is showcased in Table 3.1.

An essential feature of data lakehouses is enabling ACID transactions. In a lakehouse, ACID transactions are implemented through a transactional metadata layer built on top of an open-format filesystem in the cloud[2][12]. The system creates a transaction log, which is maintained to track table modifications such as file additions and removals[1]. ACID qualities are also supported by optimistic concurrency control and snapshot isolation, which prevent possible conflicts and consistency errors from simultaneous usage[1][7]. ACID transactions provide notable advantages to machine learning by enforcing data reliability and quality[1][2].

The metadata layer of the lakehouse also enables time traveling to historical snapshots of data. This capability provides a way to travel back to earlier states of datasets for debugging and monitoring changes. The transaction log stores each transaction with a monotonically increasing version number, which can be queried to reconstruct a consistent snapshot of the data as it existed at that specific point in time[1][7]. The transaction logs are conveniently stored in the same place as the data, which eliminates the need for a separate external storage system or database specifically meant for technical metadata[2]. Time travel capabilities support reproducibility, accelerate debugging, and provide enhanced regulatory compliance through auditability[1][30].

Another important feature of the data lakehouse architecture is schema enforce-

ment. Schema enforcement ensures that data being uploaded or written to a table matches a predefined structure such as data types, column names, and constraints[2]. Data lakehouses provide a more flexible approach to schema enforcement than traditional data warehouses by allowing schema evolution, which allows the modification of a table's structure without breaking the pipeline[6]. Schema enforcement is efficient in eliminating training-serving skew by ensuring that feature engineering logic remains consistent. Another benefit of schema enforcement is the reduction of data quality failures, which are a major part of failures in machine learning projects[1].

Data lakehouses also offer feature stores, which are designed to manage, store, and serve machine learning features across both training and inference contexts. They serve as a centralized repository for reusable features, which are accompanied by metadata information that improves feature discovery significantly[1][2]. Feature stores typically consist of two different layers, an offline store and an online store. The offline store is used for retrieving historical feature values for model training, while the online store is used for retrieving the latest feature values during inference[31]. Feature stores are also useful in preventing training-serving skew and enhancing the machine learning process by improving productivity and feature reuse[30][12].

Pipeline orchestration is an essential part of data lakehouse systems and very relevant with machine learning workloads. It involves managing the flow of data from ingestion to transformation and finally to analytics, ensuring that tasks are executed in the expected order[11]. An orchestrator defines the sequence of tasks, manages dependencies between them, handles scheduling and triggering, and deals with failures through retries and alerts[11][12]. It can be conveniently used for automating the machine learning lifecycle while also benefitting overall observability of the workflow[1].

Lakehouse feature	**MLOps principle**	**Role in the ML lifecycle**
ACID transactions	Data reliability, Data quality	Provides consistent, isolated reads and writes when multiple jobs or users update the same table
Time travel	Reproducibility, Auditability	Allows querying or restoring earlier table states to reproduce training runs, debug pipelines, and audit historical data.
Schema enforcement	Data quality, Reproducibility	Validates incoming data against the table schema and provides controlled paths for schema evolution as needed.
Feature stores	Collaboration, Reproducibility	Centralises feature definitions and serves the same values for offline training and online inference.
Pipeline orchestration	Automation, Observability	Coordinates multi-step ML pipelines with scheduling, retries, and run visibility.

Table 3.1: Lakehouse features and their role in the ML lifecycle

3.3 Feature evaluation

The lakehouse features are evaluated in this thesis by examining platform documentation and utilizing the features on the platforms in the case study to create a simple machine learning pipeline. Additional testing experiments are also conducted to observe the mechanisms and behavior of the features sufficiently. The research aims to explore how each feature is implemented in the lakehouse system and what kinds of approaches there are for utilizing the features for machine learning workloads, which means that the observed qualities are qualitative in their nature. Quantitative metrics were also considered but ultimately dropped due to restrictions of free-tier usage plans of lakehouse platforms and the overall scope of the project. The evaluation

framework is shown in Table 3.2.

When evaluating ACID transactions related to machine learning workloads, one relevant mechanism is the concurrency control approach enforced by the platform's table format. An example of this is when two pipelines write to a table that is also simultaneously being read by a training job. The concurrency control approach of a lakehouse platform can be assessed by characterizing it with established terminology such as optimistic and pessimistic concurrency control, and how possible conflict behavior is handled[1][7]. This behavior is examined with a combination of a documentation review and hands-on testing with concurrent operations to gauge the implementation of concurrency control and the resolution of conflict situations.

Time travel functionalities in data lakehouses are also implemented in various ways. The essential goal is largely the same: the reconstruction of a table's prior state as it existed at an earlier point in time. The variance in approach occurs in the way that time travel functionality is implemented in the system, how the data is referenced, and what kind of versioning methods there are[6]. In the case study, time travel capabilities are assessed through documentation review and hands-on recovery testing, building a view of how each platform utilizes versioning and how they revert a table's state to an earlier one.

To assess schema enforcement, general strictness and schema evolution capabilities can be evaluated to gather an overview of how data quality is ensured in data lakehouse environments. Altered column types, newly introduced null values, and row additions can often cause pipeline failures and quality degradation[30]. Enforcement strictness and evolution handling is assessed with a combination of experimental testing throughout the implementation of the machine learning pipeline by joining two mismatching tables together to examine enforcement and evolution possibilities.

The integration of feature stores varies significantly across lakehouse service

providers. Some are natively built in, some delegated to external tools, and others simply utilized via add-ons[1]. This integration mechanism shapes the deployment footprint, from libraries to managed services with separate offline and online stores. The consistency between these stores and the mechanism for refreshing online values are central to preventing training-serving skew. These aspects are assessed through documentation review and hands-on exploration of feature stores on each platform.

Pipeline orchestration ties the whole machine learning process and its stages together. Successful pipeline orchestration morphs individual manual steps into an automated and repeatable workflow[1][30]. Platforms differ in approaches to implement pipeline orchestration and how scheduling, dependencies, and parameterization are exposed to the user. Responsibility of these tasks is also divided differently between the engineer the platform, which is observed in the comparison. Additionally, failure handling behavior is observed across the platforms to determine how failure information reaches the engineer and how the orchestrator supports the diagnosis and debugging of the situation.

Lakehouse feature	**Examined mechanism**	**Examined behavior**
ACID transactions	Concurrency control	Write outcome
Time travel	Versioning	Recovery operation
Schema enforcement	Default strictness	Evolution handling
Feature stores	Integration	Consistency
Pipeline orchestration	Responsibility	Failure handling

Table 3.2: Examined mechanisms and observed behaviors used for evaluating lakehouse features

4 Case study

In this chapter, an open source lakehouse stack is built from individual components and compared against two managed platforms, Databricks and Snowflake, to gather insight into how machine learning is enabled across different lakehouse implementations. The dataset, model, and task used across all three platforms are the same to emphasize platform specific details and constraints. The features and evaluation axes selected in the last chapter are used in the comparison to assess how machine learning workloads are supported on each platform and how the approach to implementing a lakehouse architecture differs among them. The platforms are implemented in the following order: open source stack, Databricks, and Snowflake. Implementing the open source stack first ensures familiarity with the underlying components and the architecture itself, and provides insight on what managed platforms hide behind their interfaces.

4.1 Setup

As briefly mentioned earlier, a simple machine learning pipeline is set up and used on all three platforms to assess their qualitative aspects, differences, and effect on the machine learning lifecycle. The process for selecting the dataset and model for the task is heavily tied to the chosen features and metrics defined earlier to ensure that the pipeline is complex enough for sufficient analysis. Implementability without platform-specific data engineering and the reproducibility of the machine learning

task were also considered when selecting the dataset and model.

4.1.1 Data understanding

The dataset chosen for the machine learning pipeline is the New York City Taxi and Limousine Commission's (TLC) Yellow Taxi trip records from December 2024 to January 2025. The TLC Trip Record Data captures attributes such as pickup and drop-off times, locations, distances, fares, rate types, payment types, and passenger counts[32][33]. Each record represents one trip, and the column groups indicate the aforementioned data points. The data is published monthly in Parquet files, which is the de facto open columnar format used in data lakehouse environments and the platforms examined in this chapter. The dataset is also publicly available, well documented, and widely used in case studies which makes it a solid candidate for the platform comparison[34], [35]. The observable months were selected because of a schema change that occurred between the selected months. From January 2025 onwards, the TLC added a new column, cbd_congestion_fee, to represent a per-trip charge related to a congestion toll introduced in Manhattan in January 2025[36]. This schema change is critical in assessing schema enforcement by examining schema evolution handling and time travel capabilities across all three platforms.

The goals of the data understanding and data preparation phases are almost identical on all three platforms. Sampling is used to limit the size of the dataset and to enable reproducibility with a set seed. Both months are sampled down to 15%, which equates to about 500,000 rows per month and then ingested to the same bronze table to test schema enforcement and possible errors caused by the mismatched schemas due to the added column in January's data. The combined table of approximately one million rows of data is then cleaned and filtered to remove outliers and null values. A new feature, `trip_duration_minutes`, is derived from the two timestamps of pickup and drop off times to be used as the foundation

for the classification task. Trips lasting over 4 hours and near-zero minutes, are removed due to their inconsistency with normal taxi service patterns. The same reasoning is also applied to the removal of trips where the distance is near-zero or over 100 miles. Rows with refunds, near-zero fares, and zero passengers are also removed from the dataset. The cleaned data is then ingested to a silver table. The data preparation phase removes about 16% of the rows, making the total row count in the silver table approximately 900 thousand. Four newly derived features are also introduced to the silver table: `pickup_hour`, `pickup_dayofweek`, `is_weekend`, and `is_airport_trip`. These features capture patterns that directly affect trip duration such as rush hour, differences in weekday versus weekend traffic, and airport trips. Platform-specific methods and mechanisms for cleaning the data, handling the schema change, and performing other data preparation tasks are described later on in this chapter.

4.1.2 Task and model

The prediction task used in the pipeline is binary classification for predicting whether the trip's duration will exceed the median trip duration of the December 2024 training sample. The trip duration is calculated with the difference between `tpep_pickup_datetime` and `tpep_dropoff_datetime`, and the median threshold is determined once from the training data, which will stay the same through training and inference. The target variable, `is_long_trip`, takes the value 1 when the trip duration is over the median and 0 otherwise. The task is intentionally simple, because its purpose is mainly to exercise and assess the defined lakehouse features.

The classifier used is a logistic regression model, which was also chosen due to its simplicity and implementability on all three platforms. Logistic regression is a supervised machine learning algorithm that learns a weight for each input feature and combines them to produce a probability between 0 and 1 for the binary outcome.

Predictive performance is measured by two metrics evaluated on the January 2025 data. Accuracy measures the proportion of correctly classified trips out of all predictions. AUC, the area under the receiver operating characteristic curve, measures the model's ability to distinguish between the two classes across all classification thresholds, with 1.0 representing a perfect classifier and 0.5 representing random guessing[37].

4.2 Open source lakehouse implementation

The open source lakehouse implementation designed in this thesis consists of seven major components: object storage, table format, metadata service, compute engine, experiment tracker, feature store, and pipeline orchestrator. Each component has a specific role in the creation of a working data lakehouse environment, and it is also vital to ensure that each component is compatible and able to communicate with each other. The implementation is built with Docker, which is a popular tool for containerizing applications to separate them from the infrastructure[38]. Docker Compose is a tool for managing applications consisting of multiple containers[38]. In this implementation Docker is utilized to host the components as containers which are coordinated by Docker Compose to start, network, and persist state from a single configuration file.

4.2.1 Component overview

The entire stack runs locally via Docker Compose, with each component as a separate containerized service on a shared network. The stack can be conveniently built with the command `docker compose up -d` and torn down with `docker compose down`. The configuration file `docker-compose.yml` defines and configures the containers, including their images, networks, volumes, ports, and environment variables in a

single YAML file.

MinIO, which is a high-performance open source object storage solution, serves as the object storage component due to its S3 compatibility and seamless integration with other lakehouse-adjacent tools[39]. It serves as the location for table data and their metadata files, which are stored as objects in S3-compatible buckets. The buckets and their contents can be viewed through the MinIO console, which is accessible through a web browser.

Apache Iceberg is a reliable open table format for large analytic dataset usage, which was selected as the table format component because of its implementation of lakehouse features such as ACID transactions, schema enforcement, and snapshot-based time travel capabilities[40]. It sits between raw files in MinIO's object storage and the compute engine to apply the aforementioned table functionalities to raw data files.

The Iceberg library needs a catalog to manage and keep track of tables and their metadata[41]. The Iceberg REST catalog serves as the metadata service component, which maps table names to metadata files and snapshots located in MinIO. It serves as the bridge between the tables and their metadata by returning a pointer to the compute engine to link everything together.

Apache Spark is an analytics engine fit for processing large datasets[42]. It serves as the compute engine component, which performs tasks such as reading, writing, and machine learning runs. Spark additionally hosts Python clients for the experiment tracker and feature store components while also providing a Jupyter server for effective notebook usage through a web browser. Spark also has a machine learning library that provides common machine learning models, such as the logistic regression model used in the classification task, natively[42].

Experiment tracking is conducted via MLflow, which is an open source platform meant to assist machine learning practitioners with handling complex machine learn-

ing workloads[43]. It tracks training run records and their parameters, metrics, and output artifacts to centralized locations that can be easily queried as needed. In this implementation, tracking data including runs, parameters, and metrics are located within a relational backend while the artifacts are stored in the object storage at MinIO.

Feast is an open source feature store for managing and serving machine learning features consistently between training and inference[44]. Feast is used as the feature store component for managing feature definitions and serving the same values consistently to training through the offline store and to inference through the online store. In this implementation, feast is used as a Python library rather than as a deployed service to reduce overall complexity of the architecture.

Pipeline orchestration is conducted with Apache Airflow, which is an open source platform used for scheduling and monitoring workflows[45]. Airflow coordinates the multi-step pipeline as a DAG with scheduling, per-task retries, and a user interface accessible through a web browser for inspecting runs. The DAG is defined as a Python file, with tasks built from operators.

4.2.2 Data preparation

The December and January parquet files were retrieved from the TLC's publications and loaded into a data directory accessible to Spark. Jupyter notebooks were created for both months to perform sampling individually, and to load the sampled data to a bronze table separately to test schema enforcement. The sampled December data was ingested into a bronze table with a single Spark command. The created bronze table can be queried with a unique snapshot identifier that was assigned to it during its creation to verify the success of the ingestion. It can also be seen in the object storage through MinIO's console.

The sampled January data was ingested to the same bronze table, which caused

an error due to an expected schema mismatch due to the extra column introduced in January's data. The first try used a plain append function, which caused Spark and Iceberg to throw a schema mismatch error. Two approaches for schema evolution were tested to fix the error. The first one is a writer-driven solution, where a `mergeSchema=true` parameter was passed to the command to instruct it to widen the target schema with the new column. The second option was to manually alter the schema of the table by adding the column in with an SQL command. Both options were viable for the task and provided successful results.

During the testing of schema enforcement and evolution, an extra append landed on the bronze table by accident, making the total row count approximately 1.60 million instead of the expected 1.07 million. This mistake provided an excellent opportunity to test the snapshot-based time travel capabilities of Iceberg. The snapshot table can be queried directly via SQL with Spark to find a specific snapshot, which can be used to roll back to an earlier version of the bronze table. With a single catalog call the extra append was rolled back, making the bronze table ready for data cleansing.

Before continuing onwards with data cleansing, concurrency testing was conducted to test the validity of Iceberg's ACID transactions. The test included two writers performing appends to the same Iceberg table concurrently from threads in a single Spark driver. A specific table was created for the test, which was left in place after the run to maintain the snapshot history. The expected outcome was for one of the threads to succeed due to the implementation of optimistic concurrency control of Iceberg. However, neither of the threads succeeded due to the implementation of the REST catalog's bundled SQLite backend. Even though the outcome was different than expected, ACID principles were still adhered to.

The data cleansing was also done within a Jupyter notebook. The quality filters defined in data understanding were applied to the data, which lead to the row count

decreasing to approximately 900,000. The cleansing process is visualized in Table 4.1. The derived features mentioned in the data understanding phase were added as columns to the silver table in addition to the cleaned data, resulting in a finished silver table ready for modeling.

Step	Rows	Removed
Start (bronze)	1,073,869	
`trip_duration_minutes` $\in (1, 240]$	1,059,921	13,948
`trip_distance` $\in (0, 100]$	1,043,776	16,145
`fare_amount` > 0	1,013,849	29,927
`passenger_count` $\in [1, 6]$, non-null	900,450	113,399

Table 4.1: Row counts at each step of the open source implementation's bronze-to-silver filter chain

4.2.3 Modeling

Two of the components mentioned in the stack overview were implemented in this phase. The first one, MLflow, was set up before training to track the runs and artifacts produced in this phase. MLflow was added as a service to the `docker-compose.yml` file and tested with a separate Jupyter notebook to ensure that it was working properly with the rest of the stack. After successful testing, MLflow was ready to start tracking the upcoming training runs and their results.

The second one, Feast, was set up to introduce two extra aggregate features computed from December's data for the second iteration of training to be served consistently both at training and inference. The aggregate features were calculated in a separate Jupyter notebook by calculating two averages from December's data: average trip duration grouped by pickup hour, and average trip duration grouped by pickup zone. Both of these features were saved by Iceberg into MinIO's object storage with distinct timestamp columns. A `feature_repo` folder was also created with a configuration file and a definition file to communicate the location and description of the features for Feast. The command feast apply was then run to register the

information into Feast's own registry, and another command feast materialize was run to read and copy the values into the online store.

The first iteration of training was done on the cleansed data located in the silver table. The data was split by pickup date into separate months once again, with December data purposed for training and January data for evaluation. The median trip duration, which was 13.7 minutes, was calculated and saved into MLflow from the December training data to be applied to both months of data to derive the target feature `is_long_trip`. December's median is used for January as well to avoid a leakage situation, where January's data would see its own median. The features used were then assembled into a feature vector for Spark MLlib, which was used to train the logistic regression model with multiple iterations. The trained model was run on both the December and January sets to get predicted labels, and AUC and accuracy scores were also computed on both sets of data to detect overfitting. All the necessary results and generated artifacts such as confusion matrices were saved via MLflow for evaluation.

The second iteration of training utilized Feast and the generated aggregate features. The data was prepared in the same fashion as in the first iteration, but this time Feast was used to incorporate the new features into the training procedure. The model was trained with the two additional features, evaluated on both sets again, and everything worth tracking was also logged into MLflow.

4.2.4 Evaluation

The results of both training run iterations were fetched from MLflow and compared to each other. The results can be seen in Table 4.2. Both runs performed well, but an interesting note is that the evaluation AUC was higher than the training AUC, which is usually not the case. The addition of the aggregate features improved the results slightly, which was expected. The training runs can be re-executed against

the exact same state of the tables with snapshots, which is an important factor regarding reproducibility.

Run	Train AUC	Train Acc	Eval AUC	Eval Acc
silver-only	0.891	0.811	0.928	0.851
with-features	0.908	0.826	0.934	0.851

Table 4.2: Training and evaluation metrics for the two training iterations in the open-source lakehouse implementation

4.2.5 Deployment

Airflow was implemented as the pipeline orchestrator in this phase to wrap everything together for automation and overall observability. Up until Airflow, every Jupyter notebook was run manually one by one, which is inefficient and cumbersome. Airflow orchestrated the entire workflow from ingesting raw data all the way to training the model with a single button press from its web UI. It also logged every step in the process, which made failures trivial to detect and debug.

Airflow was also added as a separate service in Docker Compose similarly to MLflow and Feast. Airflow was then given access to the Spark container, so that it could run commands to execute the notebooks. Another tool called Papermill was added to the Spark container to allow the execution of Jupyter notebooks without the need of having to open them up manually. A DAG was created to describe the pipeline as a sequence of steps, which consisted of five steps illustrated in Figure 4.1. The DAG was triggered manually from the web UI, which resulted in five successful steps and a completed automated pipeline with the same outcome as the Jupyter notebooks produced by hand.

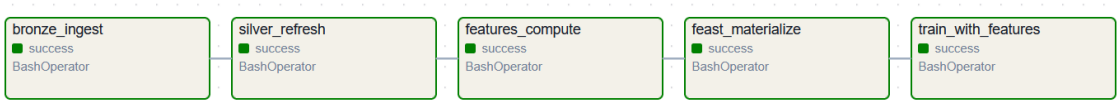

Figure 4.1: Illustration of the Airflow DAG's steps

4.2.6 Monitoring

As mentioned earlier, every training run has its metrics, parameters and model saved. Past runs can be revisited and compared, providing an overall view of model performance over time. Each Airflow run also produces logs that can be monitored for possible failures or degrades, which can be traced back to the exact step where they occur. The data itself is versioned and accessible through Iceberg snapshots, which allows the querying of past snapshots to check and verify data quality.

4.3 Databricks

Databricks is an American software company founded in 2013 by Apache Spark's creators, who are also behind the original paper introducing the lakehouse architecture[2]. Databricks is a commercially managed cloud platform built around that very architecture, with a strong background in the open source community in projects such as Delta Lake, MLflow, and Apache Spark. Most common use cases of the platform include data related tasks such as processing, storing, and analyzing data for critical business functions[46].

4.3.1 Component overview

The re-creation of the open source implementation pipeline on Databricks utilizes the same seven components, of which some tools are very similar or exactly the

same as the ones used in the open source implementation. Every component used in the pipeline runs within the Databricks workspace, which simplifies the whole setup compared to the open source setup. There is no need for a specific configuration file or container orchestration with Docker to get components to work with each other, eliminating the wiring overhead that the open source stack required. Notebooks are created in a similar fashion, which are run in the same workspace with the other services in a centralized location.

The anchor of the component setup in Databricks is the Unity Catalog, which is the metadata service and unified governance layer for data and AI assets. It replaces the Iceberg REST catalog while also absorbing the object storage layer with Unity Catalog managed storage, which is also S3-backed like MinIO. The key difference is that users interact exclusively through Unity Catalog rather than directly with the underlying buckets.

Delta Lake is the table format used in Databricks. It serves the same role as Apache Iceberg did in the open source stack, which is to provide ACID transactionality, schema enforcement, and snapshot-based time travel. Delta Lake is tightly integrated with Unity Catalog, making them a seamless package for managing storage and computing capabilities.

Databricks offers classic clusters for computing, where it is possible to configure the runtime, node types, and sizing as needed. It is also possible to use a Spark backed serverless compute option, which does the exact same thing that Apache spark did in the open source implementation. The key difference is that the other components don't need to be hosted by Spark's python clients, but instead all services live in the same workspace.

MLflow serves as the experiment tracker, but with fewer responsibilities than earlier. The artifacts produced by runs are stored and managed by Unity Catalog instead, instead of a separate MinIO bucket. MLflow experiments and runs are also

easier to find due to them being located directly in the workspace user interface, rather than a separate web service.

Databricks Feature Engineering has the same role as Feast did in the open source implementation: it manages and serves feature definitions for training and inference. The serverless compute engine does not have the feature store pre-installed, but it is installed through Python separately with a single `pip` command.

Pipeline orchestration is conducted via Databricks Workflows, which runs notebooks directly on the same serverless compute that the interactive notebooks use. This makes it more straightforward to use, since additional services and databases are not required to make it work.

4.3.2 Data preparation

The parquet files were uploaded manually from a local machine to Databricks, because the serverless compute environment restricts outbound network access from notebook cells, preventing direct download from the TLC's publication endpoint. The process was quite similar to the open source implementation: two separate notebooks were created for each month, the first one for December and the second one for January to examine schema enforcement. One thing to note is that while the same seed and sample percentage were used for sampling the data, the row counts vary slightly because of different Spark versions and input partitioning. The difference of approximately 2,000 rows is attributable to version-level differences in Spark's partition-wise Bernoulli sampling and does not materially affect the results.

A plain append function was used again to add January's data into the bronze table, which already had December's data. Delta Lake's schema enforcement rejected the write due to a metadata mismatch, which is similar to what happened with Iceberg in the open source lakehouse implementation. Writer-driven and catalog-driven evolution methods were tested similarly in Databricks, which produced the

same results.

The accidental double-append was not recreated in Databricks, but the time travel capabilities were tested by rolling back the state of the bronze table to prior to the evolution. The rollback was done with a single RESTORE TABLE command, which rolled back the table to its pre-evolution state. The rollback was confirmed by inspecting the table history and verifying that the column introduced by January's data was no longer present.

Concurrency testing was also conducted in similar fashion to the open source implementation, but this time the result was different. Instead of both threads failing, one of the threads managed to successfully perform a write operation on a test table. No partial writes were made, which means that Delta Lake's optimistic concurrency control worked as intended and upheld ACID standards.

Data cleansing for the silver table was done in a separate notebook, where the same quality filters were applied to the bronze table. The same features were also added to the silver table to ensure similarity to the silver table created in the open source lakehouse implementation. The filtering process can be seen in Table 4.3.

Step	Rows	Removed
Start (bronze)	1,071,153	
`trip_duration_minutes` $\in (1, 240]$	1,057,491	13,662
`trip_distance` $\in (0, 100]$	1,041,339	16,152
`fare_amount` > 0	1,011,732	29,607
`passenger_count` $\in [1, 6]$, non-null	898,842	112,890

Table 4.3: Row counts at each step of Databricks' bronze-to-silver filter chain

4.3.3 Modeling

MLflow was set up prior to the modeling process, which was a simple process due to the service being the workspace default option in Databricks. The experiment was created with a single call, which allowed the automatic saving of run data and

artifacts. The feature engineering library required an explicit `pip install` step because it is not pre-installed on serverless compute, though this was considerably simpler than the CLI-based Feast setup. With configuration and registry handled natively by the platform, the only remaining step was to define the feature tables in a notebook cell.

The training iterations were practically identical to the ones performed in the open source implementation. The first one used silver-only data, which was split by pickup date between the two months. The second one was performed with the two aggregate features, which were joined into the training and evaluation dataframes. All results and artifacts were logged into MLflow, which were conveniently comparable within the workspace UI.

4.3.4 Evaluation

The sampling performed in the data understanding phase altered the row counts slightly, so some deviations in AUC and accuracy scores were expected. The scores were consistent with the open source implementation within the margin expected from the minor sampling difference. The results can be seen in Table 4.4.

Run	Train AUC	Train Acc	Eval AUC	Eval Acc
silver-only	0.891	0.812	0.928	0.850
with-features	0.908	0.825	0.933	0.850

Table 4.4: Training and evaluation metrics for the two training iterations in Databricks

4.3.5 Deployment

Databricks Workflows utilized five tasks to construct the pipeline, with a slight change in order. Instead of ending the pipeline with training, a scoring step was

added to load the trained model from MLflow and write predictions to the silver schema. Each step was configured to execute a specific notebook, with each step depending on the success of the earlier step. The resulting five-task pipeline, with the added scoring step, is shown in Figure 4.2.

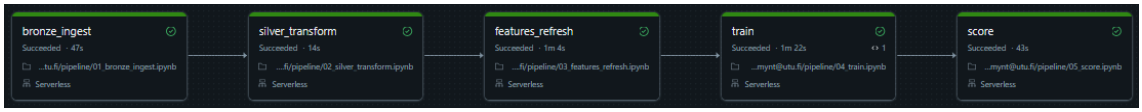

Figure 4.2: Visualization of the Databricks Workflows pipeline and its steps

4.3.6 Monitoring

MLflow and Workflows run histories and logs are found directly from the Databricks UI. Workflows offer per-task logs that can be used to determine when and where failures or delays occur in the pipeline, while Delta Lake table versioning offers the same level of auditability compared to Iceberg snapshots. Overall, the monitoring options offered by Databricks cover the same aspects as the open source lakehouse implementation.

4.4 Snowflake

Snowflake is an American cloud-native data platform that offers a hybrid data architecture built around separate storage and compute capabilities. This allows users to query data using SQL without managing infrastructure such as catalogs, clusters, or tracking servers. Snowflake originated as a cloud data warehouse, but over time has evolved to support data lakehouse workloads by offering support for most lakehouse features[47].

4.4.1 Component overview

The Snowflake lakehouse implementation fulfills the same seven roles defined for the open source and Databricks implementations, but with fewer components. Like in Databricks, everything is controlled and run within one managed platform, but in Snowflake the platform absorbs some of those roles into itself rather than separating them into separate services. There is also no need for configuration files or additional orchestration for component communication, since the services share the same managed environment together.

Unlike the open source stack, the Snowflake implementation does not deploy a separate catalog component, since it enforces the catalog's role with a hierarchy consisting of the account, database, and schema namespace to serve together. The storage component is handled by Snowflake-managed storage, which works similarly to the storage solution used in the Databricks implementation, where the buckets are never interacted with directly.

The table formatting is handled by Snowflake's native tables, which provide the same ACID qualities, schema enforcement, and time travel capabilities as the other two implementations. The biggest difference is in the format itself. The previously used Iceberg and Delta formats are open table formats layered over Parquet files, while Snowflake's native tables use its proprietary format consisting of micro-partition storage units designed to hold large quantities of compressed columnar data[48]. Snowflake also supports externally catalogued Iceberg tables, but the native format was chosen purposefully to be consistent with the usage of Delta on Databricks, and Iceberg on the open source stack.

The compute engines used in the Snowflake implementation are called virtual warehouses, which execute queries and procedures that make up the machine learning pipeline. The warehouses are managed automatically to optimize credit usage, by resuming action on user queries and suspending after idle periods. The biggest

difference to the other two implementations is the splitting of the data plane tasks with the machine learning tasks. Data ingestion, transformation, schema evolution and time travel utilize SQL, while machine learning tasks such as modeling, training, and evaluation utilize Python through the Snowpark ML library.

The experiment tracking in Snowflake is performed by Snowflake Model Registry, which fills the role of what MLflow was used for in the previous implementations. This includes the recording of trained models, their parameters, and metrics for later usage. Models and their metrics are stored and versioned within the same database in a centralized location.

The feature store role is filled by the Snowflake Feature Store, which is a similar tool to Feast and Databricks Feature Engineering. It manages and serves the feature definitions for training and inference when needed within Snowflake, without the need for separate infrastructure.

Pipeline orchestration in Snowflake is conducted through Snowflake Tasks. It chains the pipeline steps into a dependency graph, that executes the steps in order with failure handling. The key difference of Snowflake Tasks is related to how each task is run. Instead of running notebooks directly, Snowflake runs tasks as stored procedures, which are compiled database objects containing notebook functionalities.

4.4.2 Data preparation

The parquet files were manually uploaded, since they were readily available from the Databricks implementation. A separate database was created for the implementation along with a schema for each table. As in the two previous implementations, the two months of data were sampled individually with the same seed for reproducibility. The sampled December data was ingested into a bronze table, which was inferred from the schema created for it earlier. The row counts differ slightly again

despite the identical seed and sample percentage, while being only about 0.28% of difference.

The sampled January data was ingested into the same bronze table to test schema enforcement strictness once again. As expected, the insert was rejected due to the extra column, which caused a column mismatch error. The schema was evolved with a singular `ALTER` statement, which made the ingestion succeed. Snowflake also has automatic schema evolution, which can be enabled separately.

During the first append, January's data was ingested to the bronze table without sampling due to a negligent error. This allowed Snowflake's time travel capabilities to be exercised and tested by recovering the bronze table to an earlier state, where it only had December's data. Earlier states were queried by addressing a point in time through a timestamp of a state of the table. The table was recreated as a clone of the earlier state with a metadata operation. The rollback in this operation was performed similarly to the open source implementation due to the first ingestion of the January data being unsampled, leading to a faulty ingestion with too many rows of data.

Before moving on to data cleansing, concurrency testing was implemented by creating a dedicated test table for two writers to write a conflicting update to the same row. The outcome was different from the other two concurrency tests, because both writers succeeded. ACID guarantees were still upheld, because the second writer was blocked until the first one was ready. This was a direct consequence of Snowflake's concurrency control approach, which will be discussed later on in the comparison section.

The data cleansing was performed with SQL on the bronze table. The same quality filters were utilized again to remove trips with irregular durations, distances, fares, and passenger counts. The row count after the filtering process was approximately 898,000. The filtering process can be seen in Table 4.5.

Step	Rows	Removed
Start (bronze)	1,070,808	
`trip_duration_minutes` $\in (1, 240]$	1,057,083	13,725
`trip_distance` $\in (0, 100]$	1,040,965	16,118
`fare_amount` > 0	1,011,192	29,773
`passenger_count` $\in [1, 6]$, non-null	898,150	113,042

Table 4.5: Row counts at each step of Snowflake's bronze-to-silver filter chain

4.4.3 Modeling

As mentioned earlier, Snowflake's Model Registry filled the role of MLflow as the experiment tracker, which was responsible for saving run metrics, artifacts, and other relevant info about the machine learning process. There was no need for a separate setup for the experiment tracker because the registry is part of the database itself. Snowflake Feature Store did not require any setting up either except for a schema which is created automatically when instantiating a feature store.

The two training iterations followed the same structure as the open source implementation. The first iteration was trained on the six silver features, and the second one added the two aggregate features served from the feature store. The model was trained through Snowpark ML with regularization disabled to match Spark MLlib, and both training runs were logged to the registry.

4.4.4 Evaluation

The sampling differences affected the results slightly similarly to the Databricks implementation. The results were still very close to both results of the earlier implementations. The results are shown in Table 4.6. The results of the second training run with the additional aggregate features are slightly better as well, mirroring the results of the earlier implementations.

Run	Train AUC	Train Acc	Eval AUC	Eval Acc
silver-only	0.891	0.812	0.928	0.850
with-features	0.908	0.826	0.934	0.852

Table 4.6: Training and evaluation metrics for the two training iterations in the Snowflake implementation

4.4.5 Deployment

Snowflake Tasks was responsible for the pipeline orchestration similarly to Airflow on the open source implementation and Workflows on Databricks. Tasks were defined within the database and run on the same warehouse, with each step depending on the previous one's success. As mentioned earlier, the tasks were run as stored procedures rather than notebooks, which required a slightly different approach for implementing the pipeline. The pipeline is visualized in Figure 4.3.

Task	Status	Duration
T_01_BRONZE_INGEST	✅ SUCCEEDED	42s
T_02_SILVER_TRANSFORM	✅ SUCCEEDED	2s
T_03_FEATURES_REFRESH	✅ SUCCEEDED	15s
T_04_TRAIN	✅ SUCCEEDED	59s
T_05_SCORE	✅ SUCCEEDED	22s

Figure 4.3: Pipeline orchestration and tasks in Snowflake

4.4.6 Monitoring

Task run history and model training results can be either queried with SQL commands or viewed through Snowsight. The time travel capabilities of Snowflake satisfy data auditability requirements by offering the inspection of past table states to verify overall data quality. The monitoring options offered in Snowflake are similar to what was offered in Databricks.

4.5 Comparison

Essential lakehouse features and an evaluation framework were selected and defined in Chapter 3 of this thesis, which are used in the comparison to analyze different approaches for implementing a data lakehouse architecture and its core functionalities. The three lakehouse implementations created in this chapter are used as a basis for assessing and comparing each selected feature against the evaluation framework. For each feature, a summary table is created to revise and describe the observed mechanisms and behaviors that are used in the comparison.

4.5.1 ACID transactions

Platform	**Open source**	**Databricks**	**Snowflake**
Concurrency control	Optimistic	Optimistic	Pessimistic
Write outcome	None succeeded	One succeeded	Both succeeded

Table 4.7: ACID transactions mechanism and behavior comparison

The mechanism observed to assess ACID transactions is the concurrency control approach of each platform. The open source stack's Iceberg and Databricks' Delta Lake table formats utilize optimistic concurrency control, while Snowflake utilizes lock-based serialization, which is a form of pessimistic concurrency control[49]. Opti-

mistic concurrency control lets multiple writers proceed without blocking but checks for possible conflicts before committing. In pessimistic concurrency control conflicts are prevented entirely via locking of resources, where writers have to wait for the lock to be released by the initial operation.

In each of the three implementations, a brief concurrency test was conducted to see how two simultaneous write operations would be handled to ensure ACID qualities. In the open source implementation neither of the write operations succeeded, which was not the expected outcome due to a quirk in the SQLite-backed catalog setup of the open source stack. The expected outcome was a standard optimistic concurrency control result, where one of the writers succeeds and the other gets rejected due to conflict, which was exactly what happened in the Databricks implementation. The write outcome of the Snowflake implementation was caused by lock-based serialization, which caused the writing operations to be performed in a serialized sequence rather than parallel.

Even though the three outcomes were entirely different from each other, every one of them upheld ACID qualities in their own way, since none of them allowed any partial writes or caused lost updates. One important thing to note is that Iceberg should perform optimistic concurrency control similarly to Delta Lake, but the implementation in this thesis shifted some of the responsibility to the catalog as well.

4.5.2 Time travel

Platform	**Open source**	**Databricks**	**Snowflake**
Versioning	Snapshots	Commit-logs	Micro-partitions
Recovery	Revert	Revert	Clone

Table 4.8: Time travel mechanism and behavior comparison

Time travel capabilities of each platform were evaluated by examining the versioning mechanism to discuss how each platform represents historical state. Iceberg represents the historical state of data as snapshots by creating new metadata pointers over immutable data files instead of mutating existing data[50]. Delta Lake represents the historical state of data through an ordered transaction log, which is used to reconstruct the earlier versions of data[51]. Snowflake is more similar to Iceberg by utilizing micro-partitions, which it retains with linked metadata to be used as the historical versioning of data at different times.

The observed behavior on the three implementations was examined by performing recovery operations to gain insight into how each platform enables time travel into a past state of a table. In the open source implementation Iceberg used a rollback to roll the table back to a prior snapshot by changing the table's metadata to a prior version. This preserved the table's identity by not rewriting any data, instead mutating only the metadata. In the Databricks implementation Delta Lake restored a prior version of a table with the transaction log, which is mechanically similar to Iceberg's rollbacks. Both of these operations can be classified as revert operations, where the continuity of the table is preserved by reverting to an earlier state through metadata. Snowflake is fundamentally different, because it has no equivalent option for modifying the data directly. The Snowflake lakehouse implementation utilized cloning instead of rolling an existing table back, which created a new table with its own history. This caused the continuity of the table to change unlike the other two approaches, which preserved the continuity of existing tables.

All three options offer functional time travel capabilities, which was expected. While Iceberg and Snowflake have similar versioning mechanics, it is surprising to find that their recovery behaviors differ significantly from each other. The open source implementation's rollback experiment was more similar to the restore operation performed in the Databricks implementation, even though its versioning

principle is significantly different.

4.5.3 Schema enforcement

Platform	**Open source**	**Databricks**	**Snowflake**
Strictness	Cooperative	Cooperative	Centralized
Evolution	Proactive	Reactive	Reactive

Table 4.9: Schema enforcement mechanism and behavior comparison

Schema strictness is evaluated across the three implementations to compare where exactly the enforcement happens in the pipeline. Both the open source and Databricks implementations utilize cooperative writers, which means that enforcement is being handled at the writer level based on a trust model expecting all writers to cooperate accordingly. The client is responsible for checking that writers cooperate and throwing errors when they don't. Snowflake practices schema enforcement in a centralized manner, where the enforcing is conducted through the server engine itself. The difference between the approaches is caused by an architectural difference. Iceberg and Delta Lake are open table formats over the object storage, which means that the data files are theoretically directly reachable. This means that enforcement is heavily tied to writer discipline, while in Snowflake the data files are only reachable through the server engine.

The observed behavior related to schema enforcement in the implementations was evolution handling, which was tested in similar fashion on all three platforms by combining two mismatching schemas. Based on evolution behavior Iceberg is the outlier of the three due to its default proactive philosophy, where the schema change is defined beforehand. Delta Lake and Snowflake are reactive by default, which means that the schema change occurs in response to the arriving data. One thing to note is that all three support both directions, but the distinction between

them is their default approach to handling schema evolution.

When observing each implementation's way of performing schema enforcement and evolution handling, none of the platforms were consistently strict or flexible. This is ultimately tied down to the differences in storage and control models of the platforms. Another thing to note is that regardless of schema enforcement philosophy and evolution types, all platforms handle evolutions in a similar manner. The schema is evolved by updating only the metadata, while the old data files stay untouched by returning null values for the newly introduced columns when queried.

4.5.4 Feature stores

Platform	**Open source**	**Databricks**	**Snowflake**
Integration	Decoupled	Built-in	Built-in
Consistency	Imperative	Reactive	Declarative

Table 4.10: Feature store mechanism and behavior comparison

Feature store integration serves as the comparison mechanism to describe how the feature store relates to the rest of the platform. The open source stack includes a decoupled feature store with Feast, which is a separate system bolted on to the rest of the stack. In the managed platforms the feature stores are built-in within the services. The slight difference between the managed platforms is where the feature store is built into. In Databricks the feature store is located on top of the workspace, while in Snowflake it is native to the platform itself.

Feature store consistency refers to the responsibility for keeping the online serving consistent with the offline source data. The three implementations had similar uses for the feature stores, but the technical distinction between them is significantly different from each other. Feast's syncing is imperative, which means that it is conducted via an explicit command by the engineer. Additional tasks, such as

scheduling and monitoring are also the engineer's responsibility. The other two are more automatic in nature, but still slightly different. In Databricks the syncing is reactive, where the platform reacts to changes in the underlying table and pushes them on to the online store automatically[52]. In Snowflake the syncing is declarative, where the desired behavior is declared beforehand which the platform adheres to[53].

The most obvious difference between the implementations was related to the actual setup of the feature store itself. The open source stack required the most attention in the setup phase and with maintenance, while the managed platforms relied more on automation. The biggest difference among the three was less about capability, and more about how much of the responsibility does the engineer face in regard to feature consistency between the offline and online stores.

4.5.5 Pipeline orchestration

Platform	**Open source**	**Databricks**	**Snowflake**
Responsibility	Manual	Managed	Native
Failure handling	Programmatic	Push-based	Poll-based

Table 4.11: Pipeline orchestration mechanism and behavior comparison

The observed mechanism related to pipeline orchestration is the responsibility shared between the engineer and the platform, more specifically execution responsibility of the different phases of the pipeline. The open source implementation utilized Airflow for a manual setup of the whole pipeline all the way from deploying the orchestration service, writing the DAG files with Python, defining alerts, to managing dependencies. Databricks Workflows took most of the load off and abstracted most of the configuration beneath its surface, leaving the job definitions as the only manual part of the whole setup. Snowflake's pipeline orchestration was performed as a database

feature instead of a separate orchestration layer, which utilized SQL-based task definitions that were chained together to form a pipeline.

The observed behavior for evaluating pipeline orchestration across the implementation was platform specific failure handling. The behavior describes how failure information reaches the engineer, and how the orchestrator supports the bug fixing process. Airflow utilized a programmatic approach, where failure responses were given as code and error traces to be used to determine the issue. Databricks Workflows used a push-based approach, where failures were sent as email notifications and user interface state updates to notify where the errors occurred. Snowflake's poll-based approach required separate retrieval of failure data by querying through SQL-commands.

The three varied immensely with their approaches to defining pipelines and fixing errors. Airflow offered the most control and technical freedom of the three. Databricks offered a convenient way for detecting and fixing errors through its versatile user interface, while also removing some of the technical overhead that was present in the Airflow pipeline setup. Snowflake required the most manual effort for failure handling and debugging of the three. While hiding most of the manual setup phases behind the platform, the failure handling and error traces were difficult to find through separate SQL-queries while the platform was running. This led to a lot of trial and error attempts to fix occurring errors and configuring the pipeline to run smoothly.

5 Discussion

Three separate data lakehouse approaches were used to implement a simple machine learning pipeline following the defined machine learning lifecycle in Chapter 3. The observed differences between the platforms were feature-related implementation approaches instead of computational performance to highlight qualitative differences between the platforms. Each platform fulfilled the core functionalities of a data lakehouse architecture by adhering to ACID principles, offering time travel capabilities, enforcing schemas, utilizing feature stores, and supporting pipeline orchestration. The divergence between the platforms was mainly related to the observed mechanisms and behavior of the features.

The concurrency test conducted on the platforms produced three different observable outcomes, where the number of successful write operations changed between each implementation, while still adhering to ACID guarantees. In the open source implementation none of the writers succeeded, in Databricks one succeeded, and in Snowflake both succeeded. The open source result was not expected, as both writers failed because the catalog backend could not serialize the concurrent commits, meaning that the failure originated from a supporting component rather than Iceberg's concurrency control itself. The Databricks implementation utilized standard optimistic conflict resolution, while the Snowflake implementation showcased a different approach by using lock-based serialization to ensure the writers couldn't perform operations simultaneously. The difference between the results demonstrates that

while ACID principles are upheld in each implementation, the mechanism utilized to do so can vary drastically.

The differences found during the comparison are not random, as each platform's approach to implementing a data lakehouse architecture skews toward a certain direction related to their own architectural origin. The three platforms represent three different paths to implementing the same architecture. The open source stack is assembled from independent open source components, Databricks is a platform grown out of the data lake architecture, and Snowflake is originally a data warehouse platform. Snowflake's centralized schema enforcement, lock-based serialization, and SQL-driven platform are direct results of its data warehouse history. Databricks owes its functionality to its data lake history, which can be seen in how schema evolution is driven by the writer and pipeline orchestration is centered around notebooks. The open source stack's containerized ensemble of independent components causes peculiar behavior, as demonstrated by the concurrency test.

A pattern can be seen across the five features related to how responsibility is split between the engineer and the platform. The managed platforms absorb a lot of lifecycle decisions and configurations into the platform itself, while the open source implementation leaves them to the engineer as design choices. This can be seen in the configuration of feature stores and pipeline orchestration. The responsibility shift does not alter the amount of work done but changes its nature noticeably.

6 Conclusion

This thesis set out to examine the data lakehouse architecture, and its compatibility with machine learning workloads. Essential lakehouse features were selected and an evaluation framework was created to highlight different approaches for implementing a data lakehouse in a case study consisting of a three-way platform comparison.

The first research question was used to determine the most essential features that define a data lakehouse. Additionally, MLOps principles were utilized to highlight each feature's effect and importance on machine learning. The features selected were ACID transactions, time travel, schema enforcement, feature stores, and pipeline orchestration. The second research question focused on defining evaluation methods for assessing the lakehouse features and different approaches to implementing them across platforms. An evaluation framework was created to highlight different mechanisms and observed behaviors for each selected lakehouse feature. The third research question was answered by conducting a case study comparing three different platforms in their approach to implementing a data lakehouse architecture. An open source stack implementation was built first to understand the underlying components, which was then compared against two managed platforms, Databricks and Snowflake respectively. An identical simple machine learning pipeline was setup on all three platforms to gauge their differences across the established machine learning lifecycle in Chapter 3.

The case study showed that all three platforms fulfilled the core functionality

of a data lakehouse. Each implementation upheld ACID guarantees, offered time travel, enforced schemas, integrated a feature store, and supported pipeline orchestration. The differences between the platforms were related to more how the features were implemented instead of whether they were present in the implementation. Ultimately, this thesis demonstrated that the choice of the data lakehouse platform is less of a question related to feature availability, and more of a question related to how each platform implements the features and how much of the work is the engineer's responsibility.

The limitations in this thesis are mainly related to the scope of the project, which focused on the qualitative aspects of the observed features instead of examining quantitative metrics such as performance, scaling, and cost. The dataset, model, and task were kept simple on purpose to keep the focus on platform behavior, which meant that some elements of the full complexity of production level machine learning workloads were not fully captured. Some observed behavior was also shaped by the technical choices made in the open source implementation instead of the table format, which affected the integrity of the comparison slightly. Future work could extend the evaluation framework by utilizing quantitative metrics related to performance to apply it to a wider range of platforms and to give a better picture of the strengths and weaknesses between platforms. This could also be combined with a more demanding task with a larger dataset to see if the differences observed in this thesis still hold true under more complex machine learning workloads.

References

[1] P. K. R. Ponnamreddy, "Lakehouse architecture: Unifying data warehousing and advanced analytics in the cloud era", *Journal of Computer Science and Technology Studies*, vol. 7, no. 12, pp. 160–165, 2025.

[2] M. Armbrust, A. Ghodsi, R. Xin, M. Zaharia, et al., "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics", in *Proceedings of CIDR*, vol. 8, 2021, p. 28.

[3] M. B. Panicker, S. Mohammed, M. Sathe, N. K. Sundaram, N. Malhotra, et al., "The evolution of business intelligence from traditional bi to modern data warehousing", in *2024 International Conference on Advances in Computing, Communication and Materials (ICACCM)*, IEEE, 2024, pp. 1–6.

[4] M. Y. Santos and C. Costa, *Big data: concepts, warehousing, and analytics*. CRC Press, 2022.

[5] E. Blessing, S. Mike, and M. Saleh, "Modern Trends in Data Warehousing: Embracing Cloud-based Solutions and Real-time Analytics", working paper or preprint, 2024. DOI: 10.5281/zenodo.14926003. [Online]. Available: https://hal.science/hal-04972093.

[6] A. A. Harby and F. Zulkernine, "Data lakehouse: A survey and experimental study", *Information Systems*, vol. 127, p. 102 460, 2025.

[7] J. Schneider, C. Gröger, A. Lutsch, H. Schwarz, and B. Mitschang, "The lakehouse: State of the art on concepts and technologies", *SN Computer Science*, vol. 5, no. 5, p. 449, 2024.

[8] A. Nambiar and D. Mundra, "An overview of data warehouse and data lake in modern enterprise data management", *Big Data and Cognitive Computing*, vol. 6, no. 4, 2022, ISSN: 2504-2289. [Online]. Available: `https://www.mdpi.com/2504-2289/6/4/132`.

[9] R. Hai, C. Koutras, C. Quix, and M. Jarke, "Data lakes: A survey of functions and systems", *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 12 571–12 590, 2023.

[10] Z. Nargesian, P. Miller, and Arocena, "Data lake management: Challenges and opportunities", eng, *Proceedings of the VLDB Endowment*, vol. 12, no. 12, pp. 1986–1989, 2019, ISSN: 2150-8097.

[11] A. AbouZaid, P. J. Barclay, C. Chrysoulas, and N. Pitropakis, "Building a modern data platform based on the data lakehouse architecture and cloud-native ecosystem", *Discover Applied Sciences*, vol. 7, no. 3, p. 166, 2025.

[12] D. Sundar, Y. Jayaram, and J. Bhat, "A comprehensive cloud data lakehouse adoption strategy for scalable enterprise analytics", *International Journal of Emerging Research in Engineering and Technology*, vol. 3, no. 4, pp. 92–103, 2022.

[13] P. Gupta, N. K. Sehgal, and J. M. Acken, *Introduction to Machine Learning with Security: Theory and Practice Using Python in the Cloud*. Springer Nature, 2024.

[14] C. Janiesch, P. Zschech, and K. Heinrich, "Machine learning and deep learning c. janiesch et al.", *Electronic markets*, vol. 31, no. 3, pp. 685–695, 2021.

[15] Á. L. García et al., "A cloud-based framework for machine learning workloads and applications", *IEEE access*, vol. 8, pp. 18 681–18 692, 2020.

[16] D. P. Singh, "Cloud-based machine learning: Opportunities and challenges", *Int. J. Sci. Res. Comput. Sci. Eng. Inf. Technol*, vol. 10, pp. 264–270, 2024.

[17] Y. I. Alzoubi, A. Mishra, and A. E. Topcu, "Research trends in deep learning and machine learning for cloud computing security", *Artificial intelligence review*, vol. 57, no. 5, p. 132, 2024.

[18] P. Ogeti, N. Fadnavis, G. Patil, U. Padyana, and H. Rai, "Benefits and challenges of deploying machine learning models in the cloud", *Int. J. Intell. Syst. Appl. Eng.*, vol. 12, no. 22, 2024.

[19] D. Kreuzberger, N. Kühl, and S. Hirschl, "Machine learning operations (mlops): Overview, definition, and architecture", *IEEE access*, vol. 11, pp. 31 866–31 879, 2023.

[20] N. Hewage and D. Meedeniya, "Machine learning operations: A survey on mlops tool support", *arXiv preprint arXiv:2202.10169*, 2022.

[21] S. Kaski, "Intelligent data lakehouse and mlops pipelines for scalable predictive analytics in cloud-based enterprise systems", *International Journal of Research and Applied Innovations*, vol. 8, no. 6, pp. 13 135–13 142, 2025.

[22] M. Testi et al., "Mlops: A taxonomy and a methodology", *IEEE Access*, vol. 10, pp. 63 606–63 618, 2022.

[23] M. Zaharia et al., "Accelerating the machine learning lifecycle with mlflow.", *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 39–45, 2018.

[24] D. Sculley et al., "Hidden technical debt in machine learning systems", *Advances in neural information processing systems*, vol. 28, 2015.

[25] S. Shankar, R. Garcia, J. M. Hellerstein, and A. G. Parameswaran, "Operationalizing machine learning: An interview study", *arXiv preprint arXiv:2209.09125*, 2022.

[26] B. Karlaš et al., "Building continuous integration services for machine learning", in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020, pp. 2407–2415.

[27] A. Crespí, A.-L. Mesquida, M. Monserrat, and A. Mas, "Lifecycle models in machine learning development", *Expert Systems*, vol. 42, no. 4, e70029, 2025.

[28] S. Amershi et al., "Software engineering for machine learning: A case study", in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, 2019, pp. 291–300.

[29] S. Studer et al., "Towards crisp-ml (q): A machine learning process model with quality assurance methodology", *Machine learning and knowledge extraction*, vol. 3, no. 2, pp. 392–413, 2021.

[30] A. Aileni, "Ai/ml optimized lakehouse architecture: A comprehensive framework for modern data science", *World J Adv Eng Technol Sci*, vol. 15, no. 2, pp. 2099–104, 2025.

[31] L. Orr, A. Sanyal, X. Ling, K. Goel, and M. Leszczynski, "Managing ml pipelines: Feature stores and the coming wave of embedding ecosystems", *arXiv preprint arXiv:2108.05053*, 2021.

[32] New York City Taxi and Limousine Commission, *TLC Trip Record Data*, `https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page`, Accessed: 2026-05-10, 2025.

[33] New York City Taxi and Limousine Commission, *Data Dictionary – Yellow Taxi Trip Records*, `https://www.nyc.gov/assets/tlc/downloads/pdf/data_dictionary_trip_records_yellow.pdf`, Accessed: 2026-05-10, 2025.

[34] Databricks, *Getting Started with Databricks: Build a Simple Lakehouse Analytics Pipeline*, https://community.databricks.com/t5/get-started-guides/getting-started-with-databricks-build-a-simple-lakehouse/tac-p/89855, Accessed: 2026-05-10, 2026.

[35] Microsoft, *NYC Taxi and Limousine Yellow Dataset – Azure Open Datasets*, https://learn.microsoft.com/en-us/azure/open-datasets/dataset-taxi-yellow, Accessed: 2026-05-10, 2025.

[36] Metropolitan Transportation Authority, *Congestion Relief Zone Toll: Taxis and FHVs*, https://www.mta.info/fares-tolls/tolls/congestion-relief-zone/taxi-fhv-tolls, Accessed: 2026-05-10, 2025.

[37] T. Fawcett, "An introduction to roc analysis", *Pattern recognition letters*, vol. 27, no. 8, pp. 861–874, 2006.

[38] Docker, *Docker*, en, 530. Accessed: May 16, 2026. [Online]. Available: https://docs.docker.com/get-started/docker-overview/.

[39] MinIO, *MinIO*, original-date: 2015-01-14T19:23:58Z, May 2026. Accessed: May 16, 2026. [Online]. Available: https://github.com/minio/minio.

[40] Apache Software Foundation, *Apache iceberg: The open table format for analytic datasets*, https://iceberg.apache.org/, 2026. Accessed: May 16, 2026.

[41] Apache Software Foundation, *Rest catalog spec*, https://iceberg.apache.org/rest-catalog-spec//, 2026. Accessed: May 16, 2026.

[42] Apache Software Foundation, *Apache spark documentation*, https://spark.apache.org/docs/3.4.1/, version 3.4.1, 2024. Accessed: May 16, 2026.

[43] MLflow Project, *Mlflow documentation, version 2.18.0*, https://mlflow.org/docs/2.18.0/index.html, version 2.18.0, 2024. Accessed: May 16, 2026.

[44] Feast Project, *Feast: The open source feature store — introduction*, https://docs.feast.dev/, version 0.49.0, 2026. Accessed: May 16, 2026.

[45] Apache Software Foundation, *Apache airflow documentation: What is airflow?*, https://airflow.apache.org/docs/apache-airflow/stable/, 2026. Accessed: May 16, 2026.

[46] Databricks, *Databricks*, en, May 2026. Accessed: May 21, 2026. [Online]. Available: https://docs.databricks.com/aws/en/introduction/.

[47] Snowflake Inc., *What is snowflake?*, https://www.snowflake.com/en/data-cloud/platform/, Accessed: 2026-05-29.

[48] Snowflake Inc., *Micro-partitions & data clustering*, https://docs.snowflake.com/en/user-guide/tables-clustering-micropartitions, Accessed: 2026-06-02.

[49] Snowflake Inc., *Transactions*, https://docs.snowflake.com/en/sql-reference/transactions, Accessed: 2026-06-11.

[50] Apache Software Foundation, *Iceberg table spec*, https://iceberg.apache.org/spec/, Accessed: 2026-06-11.

[51] Delta Lake Project, *Concurrency control — delta lake documentation*, https://docs.delta.io/latest/concurrency-control.html, Accessed: 2026-06-11.

[52] Databricks Inc., *Use online tables for real-time feature serving*, https://docs.databricks.com/aws/en/machine-learning/feature-store/online-tables, Accessed: 2026-06-11.

[53] Snowflake Inc., *Working with feature views*, https://docs.snowflake.com/en/developer-guide/snowflake-ml/feature-store/feature-views, Accessed: 2026-06-11.

Appendix A Usage of generative AI

Generative AI was utilized in various ways during the writing of the thesis. The main tool used was Anthropic's Claude through its web user interface with Claude Sonnet 4.6 as the selected language model. Alternative tools used were platform specific agentic AI assistants on both Databricks and Snowflake, named Genie Code and Snowflake Cortex AI respectively. Every output provided by generative AI was thoroughly reviewed and carefully examined, before utilized in the writing process of the thesis.

The primary use case for generative AI was debugging and configuration support during the construction of the data lakehouse implementations in Chapter 4. Claude Sonnet was utilized for interpreting errors, suggesting fixes to notebook code, and helping keep note of the workflow of the open source implementation. Genie and Cortex were used for the same purpose on Databricks and Snowflake respectively.

Another use case for generative AI was to assist in LaTeX table creation. AI was used to generate LaTeX markup for test versions of tables and to assist with general formatting. The test versions were then modified to their final forms, with the values and labels set separately.

Generative AI was also used for a general language check, where the AI was prompted to point out obvious grammatic errors and typos in the text. All of the results were observed in sequence and applied only in cases where the finding was deemed to be of value in improving the overall quality of the thesis.