Cracking senior-level roles like **Data Engineering Lead**, **Data Architect**, or **Data Solution Architect** requires a well-structured study path that spans technical depth, architectural thinking, leadership skills, and system design. Here's a **comprehensive and practical roadmap** to guide your preparation.

🚀 Study Path for Senior Data Engineering / Architect Roles

1. Core Computer Science Fundamentals

- **Data Structures & Algorithms**
    - Arrays, Linked Lists, Trees, Graphs
    - Sorting, Searching, Hashing
    - Time/Space Complexity, Big-O Notation
    - Practice: LeetCode (Medium-Hard), HackerRank
- **System Design Basics**
    - Load balancing, caching, sharding, consistency
    - CAP theorem, latency vs throughput
    - HLD + LLD with use cases like YouTube, Uber, etc.

---

2. Advanced SQL and Data Modeling

- Complex SQL queries, CTEs, window functions
- Query optimization techniques
- Dimensional modeling: Star, Snowflake schemas
- OLTP vs OLAP, normalization/denormalization
- Data Vault modeling (for architect roles)

---

3. ETL / ELT & Data Pipeline Design

- Batch vs Stream processing
- Workflow orchestration tools: **Airflow, Dagster**
- Data ingestion best practices
- CDC (Change Data Capture)
- Data quality and lineage (Great Expectations, Monte Carlo)

---

4. Big Data Ecosystem

- **Hadoop, Hive, HDFS** – Fundamentals
- **Spark** (PySpark or Scala): RDD, DataFrame API, optimization, tuning
- **Kafka**: Streams, Connect, Kafka internals, offset management
- **Flink / Beam** (optional but good for solution architects)

---

5. Cloud Data Engineering (very important)

- **AWS** (or GCP / Azure):
    - S3, Glue, Athena, Redshift, EMR, Lambda, Step Functions
    - IAM, KMS, security policies
- **GCP**: BigQuery, Dataflow, Pub/Sub, Composer
- **Azure**: Synapse, Data Factory, Data Lake

✅ Learn **how to design cloud-native data platforms**.
✅ Practice **multi-layer architecture (bronze-silver-gold)**.

---

6. Data Warehousing and Lakehouse Architecture

- Traditional DWH (Teradata, Snowflake, Redshift)
- **Lakehouse Concepts**: Delta Lake, Apache Iceberg, Hudi
- ACID on data lakes
- Query engines: Presto, Trino, Athena

---

7. Data Governance & Security

- Data lineage, cataloging: **Apache Atlas**, **Amundsen**
- GDPR, HIPAA, and compliance concerns
- RBAC, column/row-level security
- Data encryption, PII masking

---

8. Infrastructure as Code (IaC) and DevOps for Data

- **Terraform, CloudFormation**
- **CI/CD pipelines** for data (dbt Cloud, GitHub Actions, etc.)
- Docker, Kubernetes basics (for deployment)

---

9. Data Solution Architecture

- End-to-end platform architecture:
    - Ingestion → Processing → Storage → Serving → Monitoring
- Trade-offs: batch vs real-time, cost vs latency, etc.
- Design case studies (see below)

---

10. Leadership & Soft Skills

- Stakeholder management
- Making build vs buy decisions
- Project estimation and execution planning
- Mentoring junior engineers
- Conducting architecture reviews

📚 Recommended Resources

Books

- *Designing Data-Intensive Applications* by Martin Kleppmann
- *Streaming Systems* by Tyler Akidau
- *Fundamentals of Data Engineering* by Joe Reis and Matt Housley
- *Cloud Data Management* by Abadi, Stonebraker (optional deep dive)

Courses

- **DataTalksClub DE Zoomcamp** (Free, full-stack)
- **Coursera Google Cloud DE / AWS DE Pathways**
- **Apache Spark + Kafka Udemy Courses**

🧠 Practice & Mock Interviews

Platforms

- **LeetCode / HackerRank** – For DSA + SQL
- **Excalidraw / Miro** – Practice designing system diagrams
- **InterviewQuery, Pramp, or gainlo** – Mock interviews

Prepare for case studies like:

- "Design a streaming platform for fraud detection"

- "Design a data platform for ride-sharing analytics"
- "Migrate on-prem Hadoop to cloud-based lakehouse"
- "Design CDC pipeline from Postgres to BigQuery"

🧩 Suggested Weekly Breakdown (8 Weeks)

Week	Focus Area
1-2	DSA, SQL mastery, Data modeling
3-4	Spark, Kafka, ETL pipeline design
5-6	Cloud platforms, DWH, governance
7	System Design + Architecture cases
8	Mock interviews, review, refine diagrams

---

2. Advanced SQL and Data Modeling

What are Data Engineering Services?

Data engineering is the designing, developing, and managing of data systems, architecture, and infrastructure to collect, clean, store, transform, and process large datasets to derive meaningful insights using analytical tools. These insights are shared with employees using data visualization dashboards. Data engineers combine different technologies, tools, apps, and solutions to build, deploy, and maintain the infrastructure.

Data engineering services are broadly classified into the following:

Azure Data Engineering

Microsoft Azure is a cloud solution with a robust ecosystem that offers the required tools, frameworks, applications, and systems to build, maintain, and upgrade the data infrastructure for a business. Data engineers use Azure's IaaS (Infrastructure as a Service) solutions to offer the required services.

AWS Data Engineering

AWS (Amazon Web Services) is a cloud ecosystem similar to Azure. Owned by Amazon, its IaaS tools and solutions help data engineers set up customized data architecture and streamline the infrastructure to deliver real-time analytical insights and accurate reports to employee dashboards.

GCP Data Engineering

Google Cloud Platform is the third most popular cloud platform and among the top three cloud service providers in the global market. From infrastructure development to data management, AI, and ML app development, you can use various solutions offered by GCP to migrate your business system to the cloud or build and deploy a fresh IT infrastructure on a public/ private/ hybrid cloud platform.

DATA MODELING:

What is data modeling?

Data modeling is the process of creating a visual representation or a blueprint that defines the information collection and management systems of any organization. This blueprint or data model helps different stakeholders, like data analysts, scientists, and engineers, to create a unified view of the organization's data. The model outlines what data the business collects, the relationship between different datasets, and the methods that will be used to store and analyze the data.

Why is data modeling important?

Organizations today collect a large amount of data from many different sources. However, raw data is not enough. You need to analyze data for actionable insights that can guide you to make profitable business decisions. Accurate data analysis needs efficient data collection, storage, and processing. There are several database technologies and data processing tools, and different datasets require different tools for efficient analysis.

Data modeling gives you a chance to understand your data and make the right technology choices to store and manage this data. In the same way an architect designs a blueprint before constructing a house, business stakeholders design a data model before they engineer database solutions for their organization.

Data modeling brings the following benefits:

- Reduces errors in database software development
- Facilitates speed and efficiency of database design and creation
- Creates consistency in data documentation and system design across the organization
- Facilitates communication between data engineers and business intelligence teams

What are the types of data models?

Data modeling typically begins by representing the data conceptually and then representing it again in the context of the chosen technologies. Analysts and stakeholders create several different types of data models during the data design stage. The following are three main types of data models:

Conceptual data model

Conceptual data models give a big picture view of data. They explain the following:

- What data the system contains
- Data attributes and conditions or constraints on the data
- What business rules the data relates to
- How the data is best organized
- Security and data integrity requirements

The business stakeholders and analysts typically create the conceptual model. It is a simple diagrammatic representation that does not follow formal data modeling rules. What matters is that it helps both technical and nontechnical stakeholders to share a common vision and agree on the purpose, scope, and design of their data project.

Example of conceptual data models

For example, the conceptual data model for an auto dealership might show the data entities like this:

1. A Showrooms entity that represents information about the different outlets the dealership has
2. A Cars entity that represents the several cars the dealership currently stocks
3. A Customers entity that represents all the customers who have made a purchase in the dealership
4. A Sales entity that represents the information about the actual sale
5. A Salesperson entity that represents the information about all the salespeople who work for the dealership

This conceptual model would also include business requirements, such as the following:

- Every car must belong to a specific showroom.
- Every sale must have at least one salesperson and one customer associated with it.
- Every car must have a brand name and product number.
- Every customer must provide their phone number and email address.

Conceptual models thus act as a bridge between the business rules and the underlying physical database management system (DBMS). Conceptual data models are also called domain models.

Logical data model

Logical data models map the conceptual data classes to technical data structures. They give more details about the data concepts and complex data relationships that were identified in the conceptual data model, such as these:

- Data types of the various attributes (for example, string or number)
- Relationships between the data entities
- Primary attributes or key fields in the data

Data architects and analysts work together to create the logical model. They follow one of several formal data modeling systems to create the representation. Sometimes agile teams might choose to skip this step and move from conceptual to physical models directly. However, these models are useful for designing large databases, called data warehouses, and for designing automatic reporting systems.

Example of logical data models

In our auto dealership example, the logical data model would expand the conceptual model and take a deeper look at the data classes as follows:

- The Showrooms entity has fields such as name and location as text data and a phone number as numerical data.
- The Customers entity has a field email address with the format xxx@example.com or xxx@example.com.yy. The field name can be no more than 100 characters long.
- The Sales entity has a customer's name and a salesperson's name as fields, along with the date of sale as a date data type and the amount as a decimal data type.

Logical models thus act as a bridge between the conceptual data model and the underlying technology and database language that developers use to create the database. However, they are technology agnostic, and you can implement them in any database language. Data engineers and stakeholders typically make technology decisions after they have created a logical data model.

Physical data model

Physical data models map the logical data models to a specific DBMS technology and use the software's terminology. For example, they give details about the following:

- Data field types as represented in the DBMS
- Data relationships as represented in the DBMS
- Additional details, such as performance tuning

Data engineers create the physical model before final design implementation. They also follow formal data modeling techniques to make sure that they have covered all aspects of the design.

Example of physical data models

Suppose that the auto dealership decided to create a data archive in Amazon S3 Glacier Flexible Retrieval. Their physical data model describes the following specifications:

- In Sales, the sale amount is a float data type, and the date of sale is a timestamp data type.
- In Customers, the customer name is a string data type.
- In S3 Glacier Flexible Retrieval terminology, a vault is the geographical location of your data.

Your physical data model also includes additional details such as which AWS Region you will create your vault in. The physical data model thus acts as a bridge between the logical data model and the final technology implementation.

What are the types of data modeling techniques?

Data modeling techniques are the different methods that you can use to create different data models. The approaches have evolved over time as the result of innovations in database concepts and data governance. The following are the main types of data modeling:

Hierarchical data modeling

In hierarchical data modeling, you can represent the relationships between the various data elements in a tree-like format. Hierarchical data models represent one-to-many relationships, with parents or root data classes mapping to several children.

In the auto dealership example, the parent class *Showrooms* would have both entities *Cars* and *Salespeople* as children because one showroom has several cars and salespeople working in it.

Graph data modeling

Hierarchical data modeling has evolved over time into graph data modeling. Graph data models represent data relationships that treat entities equally. Entities can link to each other in one-to-many or many-to-many relationships without any concept of parent or child.

For example, one showroom can have several salespeople, and one salesperson can also work at several showrooms if their shifts vary by location.

Relational data modeling

Relational data modeling is a popular modeling approach that visualizes data classes as tables. Different data tables join or link together by using keys that represent the real-world entity

relationship. You can use relational database technology to store structured data, and a relational data model is a useful method to represent your relational database structure.

For example, the auto dealership would have relational data models that represent the Salespeople table and Cars table, as shown here:

Salesperson ID	Name
1	Jane
2	John

Car ID	Car Brand
C1	XYZ
C2	ABC

Salesperson ID and Car ID are primary keys that uniquely identify individual real-world entities. In the showroom table, these primary keys act as foreign keys that link the data segments.

Showroom ID	Showroom name	Salesperson ID	Car ID
S1	NY Showroom	1	C1

In relational databases, the primary and foreign keys work together to show the data relationship. The preceding table demonstrates that showrooms can have salespeople and cars.

Entity-relationship data modeling

Entity-relationship (ER) data modeling uses formal diagrams to represent the relationships between entities in a database. Data architects use several ER modeling tools to represent data.

Object-oriented data modeling

Object-oriented programming uses data structures called objects to store data. These data objects are software abstractions of real-world entities. For example, in an object-oriented data model, the auto dealership would have data objects such as Customers with attributes like name, address, and phone number. You would store the customer data so that every real-world customer is represented as a customer data object.

Object-oriented data models overcome many of the limitations of relational data models and are popular in multimedia databases.

Dimensional data modeling

Modern enterprise computing uses data warehouse technology to store large quantities of data for analytics. You can use dimensional data modeling projects for high-speed data storage and retrieval from a data warehouse. Dimensional models use duplication or redundant data and prioritize performance over using less space for data storage.

For example, in dimensional data models, the auto dealership has dimensions such as Car, Showroom, and Time. The Car dimension has attributes like name and brand, but the Showroom dimension has hierarchies like state, city, street name, and showroom name.

What is the data modeling process?

The data modeling process follows a sequence of steps that you must perform repetitively until you create a comprehensive data model. In any organization, various stakeholders come together to create a complete data view. Although the steps vary based on the type of data modeling, the following is a general overview.

Step 1: Identify entities and their properties

Identify all the entities in your data model. Each entity should be logically distinct from all other entities and can represent people, places, things, concepts, or events. Each entity is distinct because it has one or more unique properties. You can think of entities as nouns and attributes as adjectives in your data model.

Step 2: Identify the relationships between entities

The relationships between the different entities are at the heart of data modeling. Business rules initially define these relationships at a conceptual level. You can think of relationships as the verbs in your data model. For instance, the salesperson sells many cars, or the showroom employs many salespeople.

Step 3: Identify the data modeling technique

After you conceptually understand your entities and their relationships, you can determine the data modeling technique that best suits your use case. For example, you might use relational data modeling for structured data but dimensional data modeling for unstructured data.

Step 4: Optimize and iterate

You can optimize your data model further to suit your technology and performance requirements. For example, if you plan to use Amazon Aurora and a structured query language (SQL), you will put your entities directly into tables and specify relationships by using foreign keys. By contrast, if you choose to use Amazon DynamoDB, you will need to think about access patterns before you model your table. Because DynamoDB prioritizes speed, you first determine how you will access your data and then model your data in the form it will be accessed.

Dimensional modeling:

Dimensional Data Modeling is one of the data modeling techniques used in data warehouse design. The concept of Dimensional Modeling was developed by Ralph Kimball which is comprised of facts and dimension tables. Since the main goal of this modeling is to improve the data retrieval so it is optimized for SELECT OPERATION. The advantage of using this model is that we can store data in such a way that it is easier to store and retrieve the data once stored in a data warehouse. The dimensional model is the data model used by many OLAP systems.

Elements of Dimensional Data Model

Facts

Facts are the measurable data elements that represent the business metrics of interest. For example, in a sales data warehouse, the facts might include sales revenue, units sold, and profit margins. Each fact is associated with one or more dimensions, creating a relationship between the fact and the descriptive data.

Dimension

Dimensions are the descriptive data elements that are used to categorize or classify the data. For example, in a sales data warehouse, the dimensions might include product, customer, time, and location. Each dimension is made up of a set of attributes that describe the dimension. For example, the product dimension might include attributes such as product name, product category, and product price.

Attributes

Characteristics of dimension in data modeling are known as characteristics. These are used to filter, search facts, etc. For a dimension of location, attributes can be State, Country, Zipcode, etc.

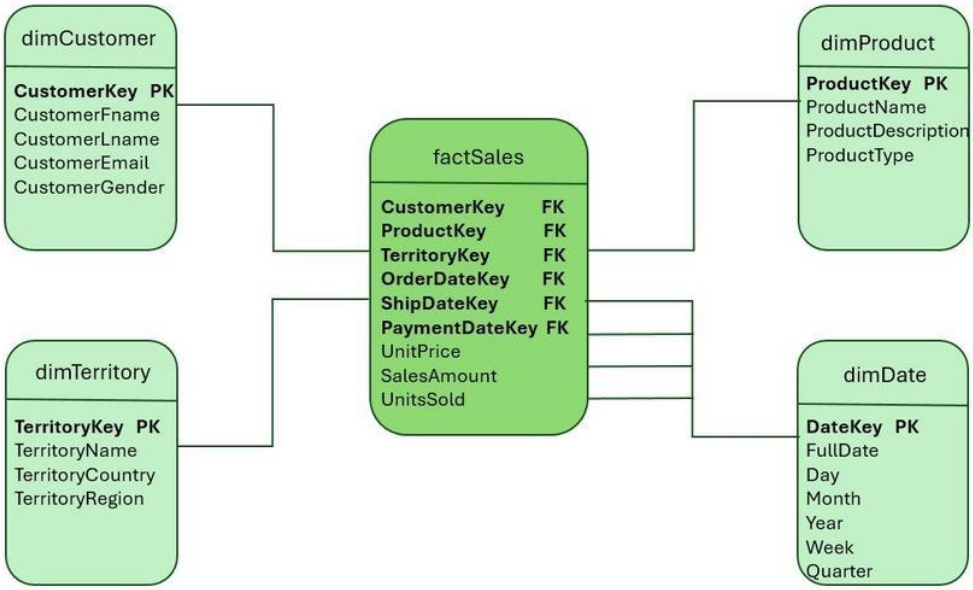
What is a Fact Table?

A fact table is a central table in the star schema of a data warehouse. It contains quantitative data for analysis and is often the largest table in the schema. The fact table records measurements, metrics, or facts about a business process. Each row in a fact table corresponds to a measurement or event and is associated with dimension tables through foreign keys.

Data Warehouse Layout

In a data warehouse layout, the fact table is surrounded by dimension tables in what is known as a star schema. This layout is designed to optimize query performance and ensure efficient data retrieval. The fact table holds the bulk of the data and is linked to multiple dimension tables, which provide context to the facts stored.

- **Star Schema:** The most straightforward and commonly used schema, where the fact table is at the center connected to dimension tables.
- **Snowflake Schema:** An extension of the star schema where dimension tables are normalized into multiple related tables.
- **Galaxy Schema:** Also known as a fact constellation schema, it contains multiple fact tables that share dimension tables.



Basic star schema Example

Types of Fact Tables

There are several types of fact tables, each serving different purposes in a data warehouse:

- **Transaction Fact Tables:** Record individual events with the highest level of detail, such as sales or order transactions.
- **Periodic Snapshot Fact Tables:** Store data at set intervals (e.g., daily or monthly) to track business performance over time, like daily inventory levels.
- **Accumulating Snapshot Fact Tables:** Track process progress across stages (e.g., order to delivery) and update as the process moves through its lifecycle.

Structure of a Fact Table

1. Quantitative Data

Fact tables primarily contain numerical data, such as sales amounts, quantities, or other metrics. This data is used for analysis and reporting.

2. Granularity

Each row in a fact table captures a single event or measurement at a specific level of detail, called **granularity** e.g., per transaction or per day.

3. Foreign Keys

Fact tables use **foreign keys** to link to **dimension tables**. For example, a sales fact table may have foreign keys for customer, product, and time, connecting to their respective dimension tables.

4. Additive, Semi-Additive, and Non-Additive Facts

- ***Additive Facts:*** These can be summed up across any of the dimensions. Example: Total sales amount.
- ***Semi-Additive Facts:*** These can be summed up for some dimensions but not all. Example: Account balances (can be summed across accounts but not time periods).
- ***Non-Additive Facts:*** These cannot be summed up across any dimension. Example: Ratios or percentages.

Importance of Fact Tables

Fact tables are critical for business intelligence and analytical processes. They enable organizations to perform complex queries and generate reports, providing insights into various aspects of the business. Here are some key functions:

Performance Measurement

Fact tables help measure the performance of business processes by storing key performance indicators (KPIs). For example, a sales fact table can track sales volume, revenue, and profit margins.

Trend Analysis

By analyzing data over time, businesses can identify trends and patterns. For example, a fact table with daily sales data can reveal seasonal trends and help forecast future sales.

Detailed Insights

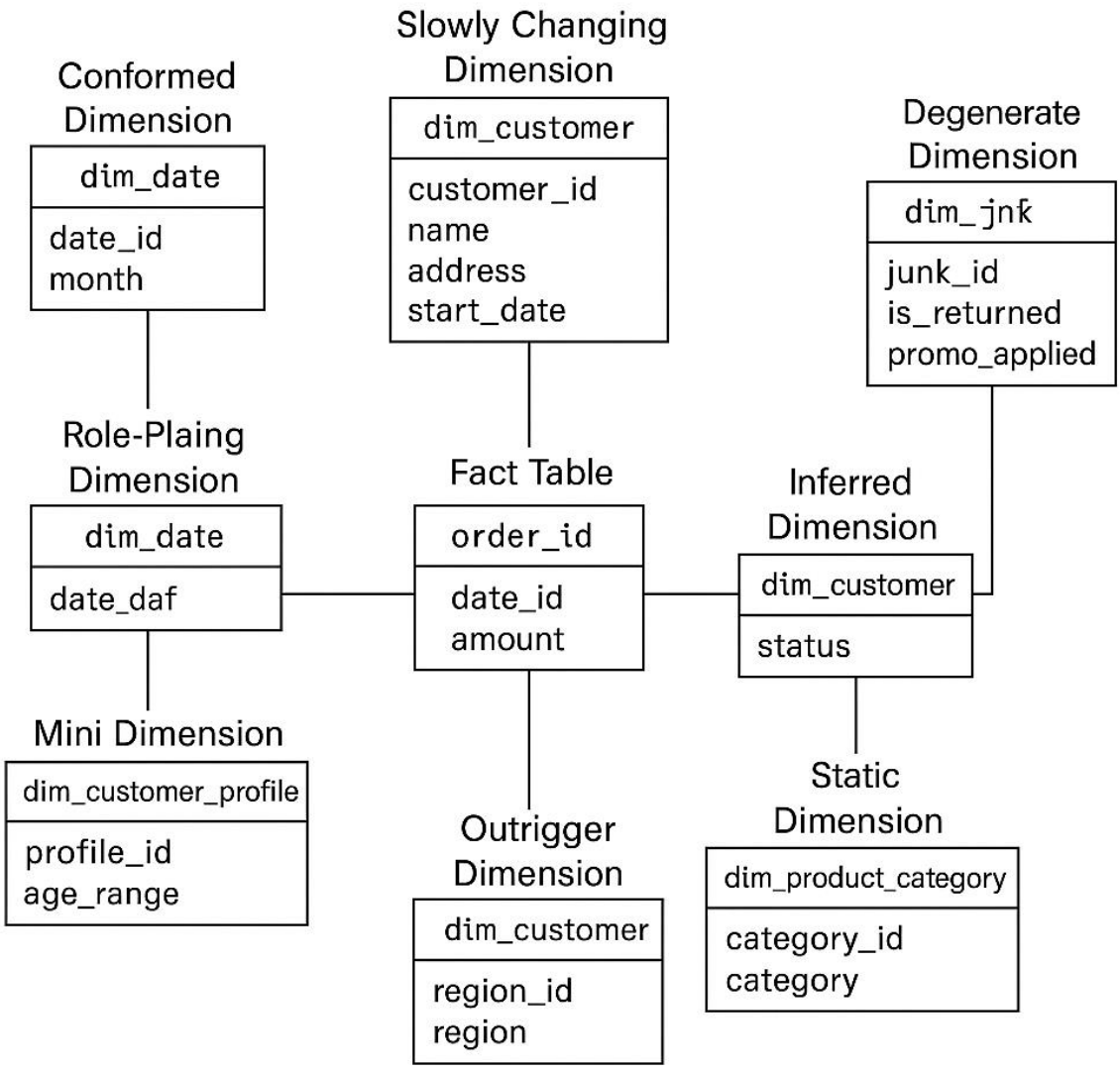
Fact tables allow for in-depth analysis at a granular level. For example, examining transaction-level data can help identify customer preferences, peak shopping times, and effective promotional strategies.

Aggregation and Summarization

Fact tables can store both detailed and aggregated data. Aggregated data helps in summarizing information for quick insights, while detailed data allows for more granular analysis.

Dimension Table

Dimensions of a fact are mentioned by the dimension table and they are basically joined by a foreign key. Dimension tables are simply de-normalized tables. The dimensions can be having one or more relationships.



Fact Table

✅ **1. Conformed Dimension**

🔷 **Definition:**

A **conformed dimension** is shared across multiple fact tables or data marts. It has the same meaning and content in all contexts where it is used.

🧠 **Purpose:**

Ensures **consistency** across reports and analytics.

📘 **Example:**

- **Date Dimension** used in both:
  - Sales fact table (sales_date)
  - Inventory fact table (stock_date)

- → Both reference the same dim_date table.

dim_date

---

date_key | date | month | quarter | year

---+---+---+---+---

20250806 | 2025-08-06 | 8 | Q3 | 2025

✅ 2. Slowly Changing Dimension (SCD)

🔷 Definition:

A dimension that **changes over time**, but not frequently.

🧠 Purpose:

To track historical changes to dimension attributes.

📘 Types & Examples:

Type	Description	Example
Type 0	No changes tracked	Customer name remains constant
Type 1	Overwrite old data	Update customer address with new one
Type 2	Track history with new row	Insert new record when address changes
Type 3	Add new column to store previous value	previous_address column

📘 Type 2 Example:

dim_customer

---

cust_id | name | address | start_date | end_date | current

---+---+---+---+---+---

101 | Alice | NY | 2020-01-01 | 2022-05-01 | N

101 | Alice | LA | 2022-05-02 | NULL | Y

✅ SCD Type 4 – History Table Approach

🔷 Definition:

Type 4 involves creating a **separate historical table** to store all the old records, while the main dimension table holds only the **current/latest data**.

🧠 **Purpose:**

To keep the dimension table lean while still maintaining full history.

📘 **Example:**

dim_customer (current data only):

customer_id	name	address	status
101	Alice	LA	Active

dim_customer_history (historical changes):

customer_id	name	address	status	start_date	end_date
101	Alice	NY	Inactive	2020-01-01	2022-05-01

✅ **Use Case:**

- Best for systems where **current data is used frequently**, and **history is queried less often**.
- Keeps query performance high on dimension joins.

✅ **SCD Type 6 – Hybrid (Types 1 + 2 + 3)**

🔷 **Definition:**

Type 6 combines the **strengths of Type 1, 2, and 3**:

- Type 1: Overwrite some attributes (e.g., status)
- Type 2: Add new row for changes (track history)
- Type 3: Keep previous value in a separate column

🧠 **Purpose:**

Provide full flexibility - **keep current, past, and historical change** tracking in one structure.

📘 **Example:**

dim_customer:

customer_sk	customer_id	name	current_address	previous_address	start_date	end_date	current_flag
201	101	Alice	LA	NY	2022-05-01	NULL	Y
150	101	Alice	NY	NULL	2020-01-01	2022-04-30	N

- previous_address: From Type 3
- new row: From Type 2
- overwrite name: From Type 1 (if changed)

Summary of SCD Types:

Type	Strategy	Tracks History?	Adds Row?	Adds Column?	Overwrites?
0	Fixed values	❌	❌	❌	❌
1	Overwrite	❌	❌	❌	✅
2	Add row	✅	✅	❌	❌
3	Add column for old value	Limited	❌	✅	✅
4	History in separate table	✅	✅ (external)	❌	❌
6	Hybrid (1+2+3)	✅	✅	✅	✅

3. Junk Dimension

🔷 Definition:

A dimension that **combines small, unrelated flags and indicators** into a single dimension table to reduce clutter in the fact table.

🧠 Purpose:

Avoid many small dimensions and keep the schema tidy.

📘 Example:

Flags like:

- is_returned
- payment_type (cash/credit)
- promo_applied

Combined into:

dim_junk

----

junk_id | is_returned | payment_type | promo_applied

--------+-------------+--------------+---------------

1 | Y | Credit | N

2 | N | Cash | Y

✅ **4. Degenerate Dimension (DD)**

🔷 **Definition:**

A dimension that **does not have its own dimension table**. It exists **only in the fact table**, typically an identifier like invoice or transaction ID.

🧠 **Purpose:**

Used for tracking at granular levels without extra tables.

📘 **Example:**

In the fact table:

fact_sales

----

invoice_id | date_key | amount

-----------+----------+--------

INV1001 | 20250806 | 250.00

No separate dimension for invoice_id.

✅ **5. Role-Playing Dimension**

🔷 **Definition:**

A dimension that plays **multiple roles** in the same database schema.

🧠 **Purpose:**

Allows reuse of the same dimension for different purposes.

📘 **Example:**

Date Dimension used for:

- order_date
- ship_date

- delivery_date

All referencing dim_date.

fact_orders

---

order_id | order_date_key | ship_date_key

---+---+---

O123 | 20250801 | 20250803

✅ **6. Inferred Dimension**

🔷 **Definition:**

A placeholder dimension record created when a fact row arrives **before the corresponding dimension row**.

🧠 **Purpose:**

Avoids dropping or delaying fact rows due to missing dimension data.

📘 **Example:**

fact_sales arrives with customer_id = 500, but no customer record exists yet.

Insert placeholder:

dim_customer

---

cust_id | name | status

---+---+---

500 | Unknown | Inferred

Later, update when customer info arrives.

✅ **7. Static Dimension**

🔷 **Definition:**

A dimension that **never changes** once loaded.

🧠 **Purpose:**

Simple lookup, no history needed.

📘 **Example:**

- dim_country

- dim_product_category

dim_country

---

country_id | name

----------+------

1 | USA

2 | Canada

✅ **8. Mini Dimension**

🔷 **Definition:**

A subset of a large dimension, often created to handle attributes that change frequently or are used in aggregate queries.

🧠 **Purpose:**

Improve performance by reducing size of frequently accessed data.

📘 **Example:**

Instead of storing customer demographics in a large dim_customer, create a dim_customer_profile mini-dimension.

dim_customer_profile

---

profile_id | age_range | income_bracket

----------+----------+---------------

1 | 18-25 | Low

2 | 26-35 | Medium

✅ **9. Outrigger Dimension**

🔷 **Definition:**

A dimension table that references another dimension table. (A dimension referencing another dimension.)

🧠 **Purpose:**

Support normalization (usually in snowflake schemas).

📘 **Example:**

dim_employee references dim_department:

dim_employee

---

emp_id | name | dept_id

---+---+---

1 | John | 10

dim_department

---

dept_id | dept_name

---+---

10 | Sales

Summary Table:

Type	Purpose	Example
Conformed	Shared across fact tables	dim_date, dim_customer
SCD	Tracks changes over time	dim_customer with history
Junk	Combines unrelated flags	is_returned, payment_type
Degenerate	Exists only in fact table	invoice_id
Role-Playing	Same dimension, multiple roles	order_date, ship_date
Inferred	Placeholder record for late dimension	Unknown Customer
Static	Never changes	dim_country
Mini	Subset for frequently changing attributes	dim_customer_profile
Outrigger	Dimension referencing another dimension	dim_employee → dim_department

Star Schema vs Snowflake Schema: Differences & Use Cases

This guide breaks down star and snowflake schemas - two common ways to organize data in warehouses. You'll learn how they work, how they're different, and when to use each to fit your data needs.

If you work with data warehouses, you know how important it is to structure data in a way that's efficient and easy to handle. But have you ever thought about which **database schema** best suits your needs? There are two major frameworks that you can use for this: **the star schema** and **the snowflake schema**.

The star schema is simple and fast - ideal when you need to extract data for analysis quickly. On the other hand, the snowflake schema is more detailed. It prioritizes storage efficiency and managing complex data relationships.

What is a Star Schema?

A star schema is a way to organize data in a database, especially in **data warehouses**, to make it easier and faster to analyze. At the center, there's a main table called the **fact table**, which holds measurable data like sales or revenue. Around it are **dimension tables**, which add details like product names, customer info, or dates. This layout forms a star-like shape.

Let's look at the key features of the star schema:

- **Single-level dimension tables**: The dimension tables connect directly to the fact table without extra layers. Each table focuses on one area, like products, regions, or time, making it simple to use.
- **Denormalized design**: In a star schema, related data is stored together in one table using a denormalized approach. For example, a product table may include the product ID, name, and category in the same place. While this may mean some data repetition, it processes queries faster.
- **Common in data warehousing**: The star schema is used for quick analysis. It can easily filter or calculate totals, so it's likely a good choice for data warehouses where fast insights are required.

Let's understand this with a simple star schema diagram. The **fact table** Sales is in the center. It holds the numeric data you want to analyze, like sales or profits. Connected to it are **dimension tables** with descriptive details, such as product names, customer location, or dates:

However, you would have to accept some data redundancy since the dimension tables may contain repeated information.

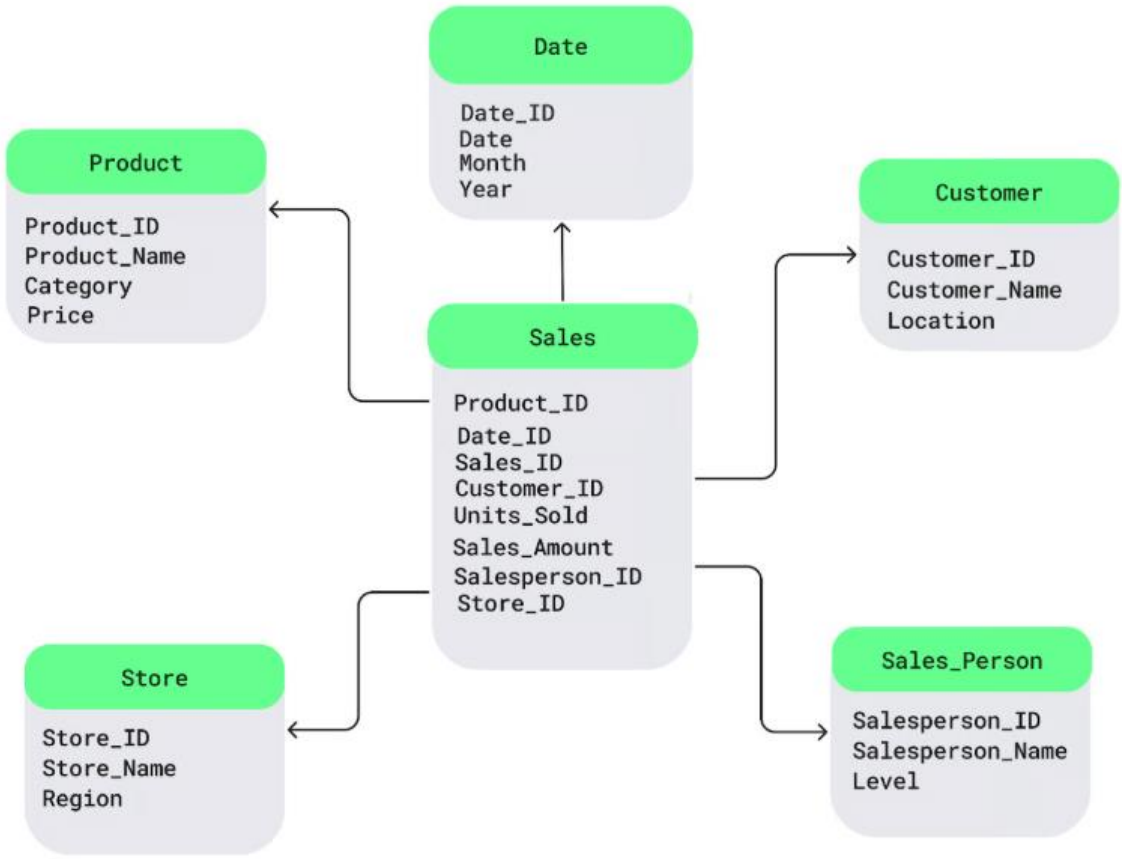
Advantages and limitations of a star schema

Now that you know what star schema is, let's look at why it stands out:

- **Faster query performance**: The star schema simplifies data retrieval by making queries fast. For example, if I want to look at sales trends, I will join the fact table with the right dimension tables. And the best part is that I will do all this without dealing with complex relationships. This would make my queries run faster and save me a lot of time.
- **Easy to understand**: Its structure is logical and simple to understand, even for non-technical users. New team members can quickly understand which tables contain the data they need, speeding up analysis and simplifying maintenance.

Despite all the benefits, star schema does have a drawback. As I mentioned before, due to denormalization, dimension tables often contain repeated information, which **increases**

storage use. For example, if several products belong to the same category, each product's name might repeat, taking up more storage space.



What is a Snowflake Schema?

A **snowflake schema** is another way of organizing data. In this schema, dimension tables are split into smaller sub-dimensions to keep data more organized and detailed - just like snowflakes in a large lake.

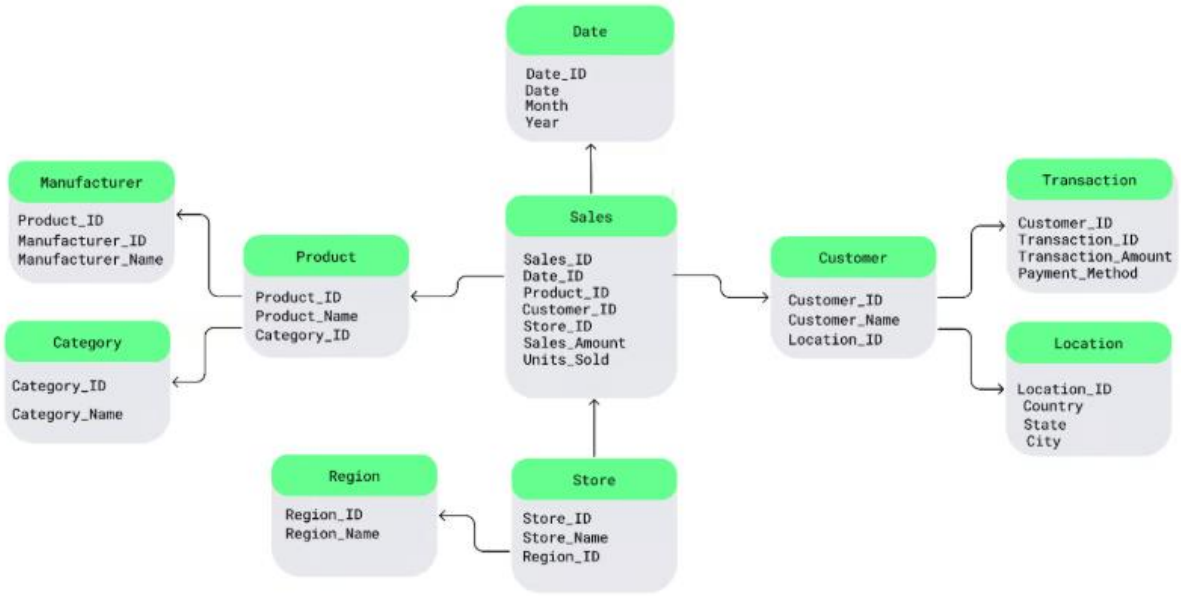
Let's look at the key features of the snowflake schema that make it different from other schemas:

- **Multi-level dimension tables**: We can break down our dimension tables into smaller, more specific tables. For example, if I want to track store locations, instead of putting all location details in one big table, I can split them into separate tables for countries, states, and cities. This way, each table would contain only the information it needs to reduce redundancy and improve organization.
- **Normalization for storage efficiency**: Unlike star schema, the snowflake schema follows a **normalized design**, which avoids data duplication. For example, rather than repeating a product category like Electronics for every product, I can store the category in a separate table and link it to individual products.

- Suitability for complex data environments: The snowflake schema works best for complex data environments because it uses multi-level tables to handle intricate relationships and hierarchical data structures.

Let's understand this with a simple snowflake schema diagram. At the center is the fact table, which contains measurable data. It connects to dimension tables that describe the facts, and these dimension tables further branch out into sub-dimension tables, forming a snowflake-like structure.

For example, here I split the Product into Manufacturer and Category tables and the Customer table into Transaction and Location tables:



Advantages and limitations of a snowflake schema

Like star schema, snowflake schema also has its own advantages. Let's see what they are:

- **Less data redundancy**: Normalization ensures that the same data is not stored multiple times, which reduces duplication.
- **Efficient storage for large datasets**: This schema saves storage space by avoiding repeated data, making it ideal for managing large datasets.

However, despite its advantages, there are a few limitations too. For example, **queries can be slower** because there are more joins between tables. Apart from this, the multi-level structure is more **challenging to design and maintain** than simpler schemas like star schema. So, go for it only if you have an experienced DBA team.

Differences Between Star Schema and Snowflake Schema

Both star and snowflake schemas are widely used in data warehousing, but their unique characteristics make them suitable for different needs. Let's see how these schemas differ in structure, performance, storage requirements, and use cases.

Structure

All dimension tables connect directly to one central fact table in a star schema. This means all your reference data is one step away from your main data, making it easy to understand and work with.

In comparison, a snowflake schema breaks dimension tables into smaller, more specific sub-dimension tables. For example, you can have separate tables for countries, states, and cities instead of one location table. While this creates a more organized and detailed structure, it also means more connections (or joins) are required to access your data - a primary reason why snowflake schema is more complex than star schema.

Performance

When it comes to speed, star schemas are better. Since all dimension tables connect directly to the fact table, queries require fewer joins, which means faster performance. Let's say you want to analyze sales by region - in this case, you can use the star schema to retrieve the data with minimal processing.

Conversely, Snowflake schemas are slower because you have to connect through multiple tables to retrieve the data. Each join adds processing time, making snowflake schemas less efficient for tasks that require quick query results.

Storage requirements

Star schemas take up more storage space because they store redundant information in dimension tables. For example, if multiple products belong to the same category, the category name will repeat for each product, increasing storage needs.

However, snowflake schemas normalize data to store all information only once. For example, instead of repeating category names, they are stored in a separate table and linked to the product table using foreign keys. This design saves storage space, making it ideal for large datasets.

Use cases

Star schemas are ideal for **online analytical processing** (OLAP) systems, reporting, and business intelligence tasks. Their simplicity makes them perfect for scenarios where speed and ease of use are important, such as generating quick dashboards or sales reports.

Snowflake schemas are often used for **financial analysis** or customer relationship management (CRM) systems. Organizing detailed hierarchies and saving storage space are more important than query speed in such cases.

When to Use a Star Schema

If you primarily want to organize your data simply and quickly, the star schema would be perfect. Here's when you can use it:

- If you want to run simple queries like finding total sales by region, use star schema. Since all the dimension tables connect directly to the fact table, it avoids unnecessary complexity and delivers answers faster.

- You can even use star schema when speed is your priority. It minimizes the number of table joins, so your queries run faster. I used it once to generate several sales reports, saving me so much time compared to other designs.
- If your dataset is small to medium, the star schema's redundancy won't be a problem. Even with repeated data, it'd work fine without overwhelming your storage.

When to Use a Snowflake Schema

Snowflake schema is more suitable for handling frequent updates or organizing detailed hierarchies. Here's when you can use it:

- Use the snowflake schema if you work with large datasets and want to save storage space. It normalizes dimension tables to prevent repeated data, which reduces storage requirements.
- You can even use the snowflake schema if your data changes often, like updating region names. It maintains consistent updates across all related data to minimize errors and maintenance efforts.
- If your analysis involves multiple levels of data, the snowflake schema can help you organize and represent these relationships in a simple way.

Summary Table: Star Schema vs. Snowflake Schema

Here's a quick comparison of the star and snowflake schemas to help you decide which best suits your data needs. I've highlighted the key differences in this table, focusing on their structure, performance, storage, and use cases:

Feature	Star schema	Snowflake schema
Structure	Central fact table linked to denormalized dimensions	Central fact table linked to normalized dimensions
Complexity	Simple, with fewer joins	Complex, with more joins
Data redundancy	Higher redundancy due to denormalized dimensions	Lower redundancy due to normalized dimensions
Query performance	Faster queries due to simpler structure	Slower queries because of additional joins

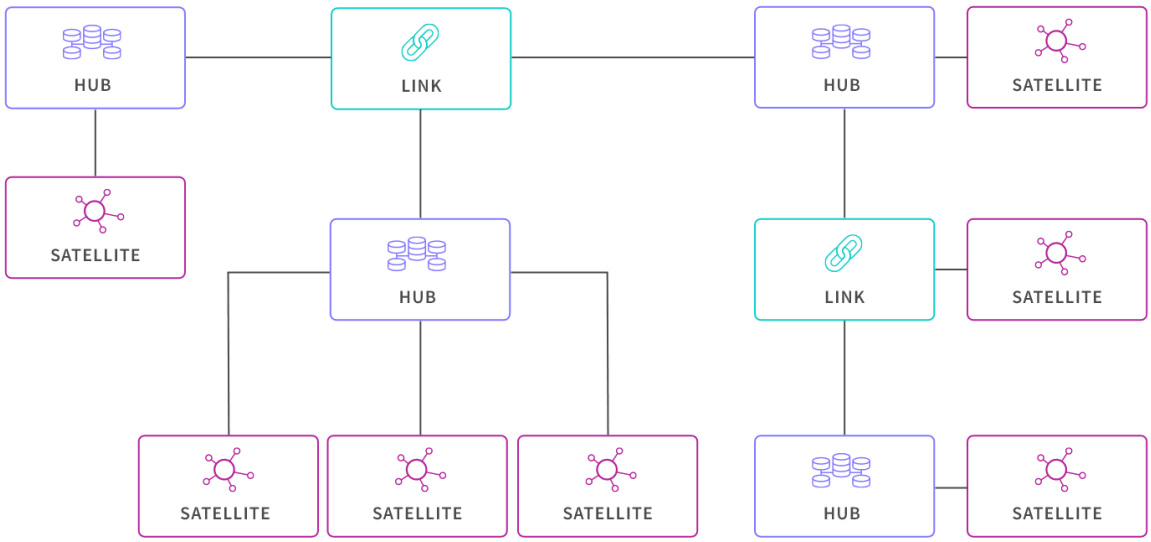
Storage	Requires more storage because of redundancy	Requires less storage due to normalization
Ease of maintenance	Easier to design and maintain	More complex to design and maintain
Best suited for	Small to medium-sized datasets	Large and complex datasets

Data Vault modelling:

A data vault is a data modeling approach and methodology used in enterprise data warehousing to handle complex and varying data structures. It combines the strengths of 3rd normal form and star schema. This hybrid approach provides a flexible, agile, and scalable solution for integrating and managing large volumes of data from diverse sources to support enterprise-scale analytics.

Data Vault Modeling

Data vaults have 3 types of entities: Hubs, Links, and Satellites.



Hubs:

- Represent core business entities like customers, products, or stores.
- Contain the business key and essential fields.

- Do not hold contextual or detailed entity information.
- Maintain a one-row-per-key constraint, ensuring that each key is associated with only one entry.

Links:

- Establish relationships between different business entities.
- Connect hubs to illustrate associations between entities.
- Help capture how entities relate to each other in a structured manner.

Satellites:

- Store descriptive attributes related to the business entities in hubs (these can change over time, similar to a Kimball Type II slowly changing dimension).
- Contain additional information like timestamps, status flags, or metadata.
- Provide context and historical data about the entities over time.

Benefits

A Data Vault (DV) provides you a robust foundation for building and managing enterprise data warehouses, especially in scenarios where data sources are numerous, diverse, and subject to change.

The benefits of using a this modeling technique include:

Flexibility: DV's are based on agile methodologies and techniques, so they're designed to handle changes and additions to data sources and business requirements with minimal disruption. This makes them well-suited for environments with evolving data requirements, such as adding or deleting columns, new tables, or new/altered relationships.

Scalability: DV's can accommodate large volumes of data (up to PBs volumes) and support the integration of data from a wide range of sources. As such, this model is a great fit for organizations implementing a data lake or data lakehouse.

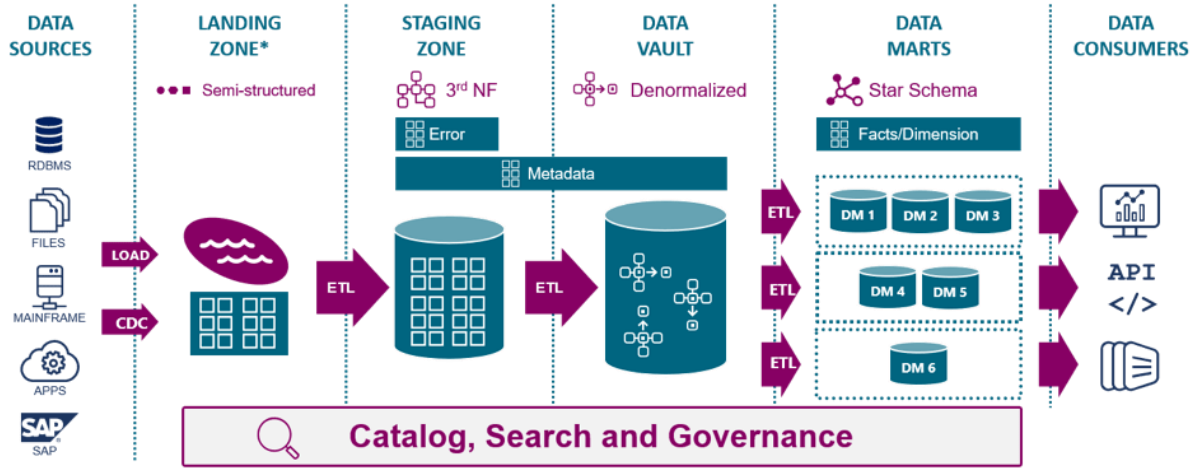
Auditability: They maintain a complete history of the data, making it easier to track changes over time and meet HIPAA compliance and other auditing requirements and regulations.

Ease of Maintenance: They simplify the process of incorporating new data sources or modifying existing ones, reducing the time and effort required for maintenance. For example, there's less need for extensive refactoring of ETL jobs when the model undergoes changes. Plus, this approach simplifies the data ingestion process, removes the cleansing requirement of a Star Schema.

Parallelization: Data can be loaded in parallel, allowing for efficient processing of large datasets.

Ease of Setup: They have a familiar architecture–employing data layers, ETL, and star schemas–so your teams can establish this approach without extensive training.

Data Vault Architecture



Let's walk through the diagram above.

- Data from source systems such as transactional, supply chain, and CRM applications is either batch loaded or streamed real-time using CDC.
- It's important for your data integration system to have robust catalog, search, and data governance capabilities to support this entire process.
- This semi-structured data is placed in a landing zone such as a data lake.
- It is then extracted, transformed, and loaded (ETL) as Inmon's relational 3rd normal form data into a staging zone (or "Raw Data Vault") repository such as a data warehouse.
- ETL is performed again to bring denormalized data into the Data Vault model.
- ETL processes are performed on an as-needed basis to load this data into star schema formatted (dimensional model) data marts.
- Data consumers can then access relevant, structured data for use cases such as analytics, visualizations, data science, and APIs to trigger alerts and actions in other systems.

Data Vault 2.0

Data Vault 2.0 is an open source extension and refinement of Data Vault 1.0. Here are key facts:

- Introduced by Dan Linstedt and Michael Olschimke in 2013
- Maintains the hub-and-spoke architecture
- Introduces additional features and best practices
    - Encourages the use of advanced technologies like big data platforms, cloud computing, and automation tools to enhance data management capabilities
- Introduces a new architecture that includes a persistent staging area, a presentation layer in data marts, and data quality services:
    - Raw vault: Contains the original source data

- Business vault: Contains business rules and transformations applied to the raw vault
- Information mart: Presentation layer providing analytical capabilities
- Data mart: Presentation layer providing reporting capabilities to end users

How a Data Vault Solves 5 Key Enterprise Data Warehouse (EDW) Challenges

Challenge #1: Adapting to constant change

DV Solution: Through the separation of business keys (as they are generally static) and the associations between them from their descriptive attributes, a Data Vault (DV) confronts the problem of change in the environment. Using these keys as the structural backbone of a data warehouse all related data can be organized around them. These Hubs (business keys), Links (associations), and SAT (descriptive attributes) support a highly adaptable data structure while maintaining a high degree of data integrity. Dan Linstedt often correlates the DV to a simplistic view of the brain where neurons are associated with Hubs and Satellites and where dendrites are Links (vectors of information). Some Links are like synapses (vectors in the opposite direction). They can be created or dropped on the fly as business relationships change, automatically morphing the data model as needed without impacting the existing data structures.

Challenge #2: Really big data

DV Solution: Data Vault 2.0 arrived on the scene in 2013 and incorporates seamless integration of big data technologies along with methodology, architecture, and best practice implementations. Through this adoption, very large amounts of data can easily be incorporated into a DV designed to store using products like MongoDB, NoSQL, Apache Cassandra, Amazon DynamoDB, and many other NoSQL options. Eliminating the cleansing requirements of a star schema design, the DV excels when dealing with huge data sets by decreasing ingestion times, and enabling parallel insertions which leverages the power of big data systems.

Challenge #3: Complexity

DV Solution: Crafting an effective and efficient DV model can be done quickly once you understand the basics of the 3 table types: Hub, Satellite, and Link. Identifying the business keys 1st and defining the Hubs is always the best place to start. From there Hub-Satellites represent source table columns that can change, and finally Links tie it all up together. Remember it is also possible to have Link-Satellite tables too. Once you've got these concepts, it's easy. After you've completed your DV model the next common thing to do is build the ETL data integration process to populate it. While a DV data model is not limited to EDW/BI solutions, anytime you need to get data out of some data source and into some target, a data integration process is generally required.

Challenge #4: The Business Domain (fitting the data to meet the needs of your business, not the other way around)

DV Solution: The DV essentially defines the Ontology of an Enterprise in that it describes the business domain and relationships within it. Processing business rules must occur before populating a Star Schema. With a DV, you can push them downstream, post EDW ingestion. An additional DV philosophy is that all data is relevant, even if it is wrong. Dan Linstedt suggests that data being wrong is a business problem, not a technical one. We agree! An EDW is really

not the right place to fix (cleanse) bad data. The simple premise of the DV is to ingest 100% of the source data 100% of the time; good, bad, or ugly. Relevant in today's world, auditability and traceability of all the data in the data warehouse thus become a standard requirement. This data model is architected specifically to meet the needs of today's EDW/BI systems.

Challenge #5: Flexibility

DV Solution: The DV methodology is based on SEI/CMMI Level 5 best practices and includes many of its components combining them with best practices from Six Sigma, TQM, and SDLC (Agile). DV projects have short controlled release cycles and can consist of a production release every 2 or 3 weeks automatically adopting the repeatable, consistent, and measurable projects expected at CMMI Level 5. When new data sources need to be added, similar business keys are likely, new Hubs-Satellites-Links can be added and then further linked to existing DV structures without any change to the existing data model.

What's the difference between OLAP and OLTP?

Online analytical processing (OLAP) and online transaction processing (OLTP) are data processing systems that help you store and analyze business data. You can collect and store data from multiple sources-such as websites, applications, smart meters, and internal systems. OLAP combines and groups the data so you can analyze it from different points of view. Conversely, OLTP stores and updates transactional data reliably and efficiently in high volumes. OLTP databases can be one among several data sources for an OLAP system.

Read about OLAP »

What are the similarities between OLAP and OLTP?

Both online analytical processing (OLAP) and online transaction processing (OLTP) are database management systems for storing and processing data in large volumes. They require efficient and reliable IT infrastructure to run smoothly. You can use them both to query existing data or store new data. Both support data-driven decision-making in an organization.

Most companies use OLTP and OLAP systems together to meet their business intelligence requirements. However, the approach to and purpose of data management differ significantly between OLAP and OLTP.

Key differences: OLAP vs. OLTP

The primary purpose of online analytical processing (OLAP) is to analyze aggregated data, while the primary purpose of online transaction processing (OLTP) is to process database transactions.

You use OLAP systems to generate reports, perform complex data analysis, and identify trends. In contrast, you use OLTP systems to process orders, update inventory, and manage customer accounts.

Other major differences include data formatting, data architecture, performance, and requirements. We'll also discuss an example of when an organization might use OLAP or OLTP.

Data formatting

OLAP systems use multidimensional data models, so you can view the same data from different angles. OLAP databases store data in a cube format, where each dimension represents a different data attribute. Each cell in the cube represents a value or measure for the intersection of the dimensions.

In contrast, OLTP systems are unidimensional and focus on one data aspect. They use a relational database to organize data into tables. Each row in the table represents an entity instance, and each column represents an entity attribute.

Data architecture

OLAP database architecture prioritizes *data read* over *data write* operations. You can quickly and efficiently perform complex queries on large volumes of data. Availability is a low-priority concern as the primary use case is analytics.

On the other hand, OLTP database architecture prioritizes *data write* operations. It's optimized for write-heavy workloads and can update high-frequency, high-volume transactional data without compromising data integrity.

For instance, if two customers purchase the same item at the same time, the OLTP system can adjust stock levels accurately. And the system will prioritize the chronological first customer if the item is the last one in stock. Availability is a high priority and is typically achieved through multiple data backups.

Performance

OLAP processing times can vary from minutes to hours depending on the type and volume of data being analyzed. To update an OLAP database, you periodically process data in large batches then upload the batch to the system all at once. Data update frequency also varies between systems, from daily to weekly or even monthly.

In contrast, you measure OLTP processing times in milliseconds or less. OLTP databases manage database updates in real time. Updates are fast, short, and triggered by you or your users. Stream processing is often used over batch processing.

Read about streaming data »

Read about batch processing »

Requirements

OLAP systems act like a centralized data store and pull in data from multiple data warehouses, relational databases, and other systems. Storage requirements measure from terabytes (TB) to petabytes (PB). Data reads can also be compute-intensive, requiring high-performing servers.

On the other hand, you can measure OLTP storage requirements in gigabytes (GB). OLTP databases may also be cleared once the data is loaded into a related OLAP data warehouse or data lake. However, compute requirements for OLTP are also high.

- **Example of OLAP vs. OLTP**

Let's consider a large retail company that operates hundreds of stores across the country. The company has a massive database that tracks sales, inventory, customer data, and other key metrics.

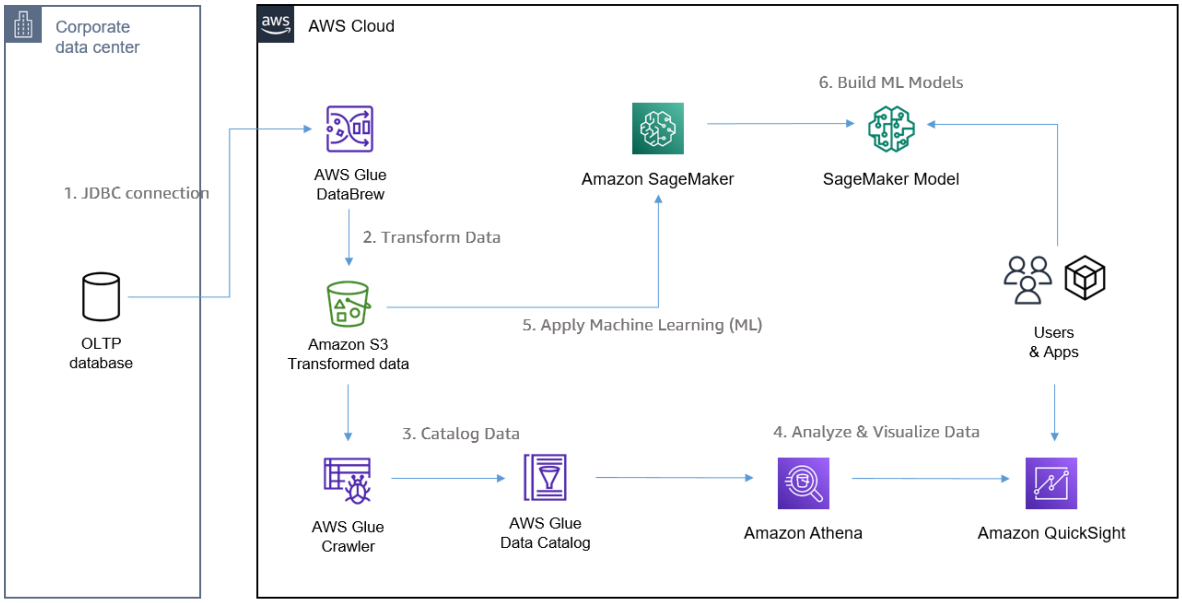
The company uses OLTP to process transactions in real time, update inventory levels, and manage customer accounts. Each store is connected to the central database, which updates the inventory levels in real time as products are sold. The company also uses OLTP to manage customer accounts-for example, to track loyalty points, manage payment information, and process returns.

In addition, the company uses OLAP to analyze the data collected by OLTP. The company's business analysts can use OLAP to generate reports on sales trends, inventory levels, customer demographics, and other key metrics. They perform complex queries on large volumes of historical data to identify patterns and trends that can inform business decisions. They identify popular products in a given time period and use the information to optimize inventory budgets.

When to use OLAP vs. OLTP

Online analytical processing (OLAP) and online transaction processing (OLTP) are two different data processing systems designed for different purposes. OLAP is optimized for complex data analysis and reporting, while OLTP is optimized for transactional processing and real-time updates.

Understanding the differences between these systems can help you make informed decisions about which system meets your needs better. In many cases, a combination of both OLAP and OLTP systems may be the best solution for businesses that require both transaction processing and data analysis. Ultimately, choosing the right system depends on the specific needs of your business, including data volume, query complexity, response time, scalability, and cost.



Summary of differences: OLAP vs. OLTP

Criteria	OLAP	OLTP

Purpose	OLAP helps you analyze large volumes of data to support decision-making.	OLTP helps you manage and process real-time transactions.
Data source	OLAP uses historical and aggregated data from multiple sources.	OLTP uses real-time and transactional data from a single source.
Data structure	OLAP uses multidimensional (cubes) or relational databases.	OLTP uses relational databases.
Data model	OLAP uses star schema, snowflake schema, or other analytical models.	OLTP uses normalized or denormalized models.
Volume of data	OLAP has large storage requirements. Think terabytes (TB) and petabytes (PB).	OLTP has comparatively smaller storage requirements. Think gigabytes (GB).
Response time	OLAP has longer response times, typically in seconds or minutes.	OLTP has shorter response times, typically in milliseconds
Example applications	OLAP is good for analyzing trends, predicting customer behavior, and identifying profitability.	OLTP is good for processing payments, customer data management, and order processing.

How can AWS support your OLAP and OLTP requirements?

Analytics on Amazon Web Services (AWS) provides various managed cloud services for online analytical processing (OLAP) and online transaction processing (OLTP) operations. From data movement, data storage, data analytics, and more, AWS offers purpose-built services that provide the best price performance, scalability, and lowest cost.

Here are examples of AWS services that can support your OLAP and OLTP needs:

- Amazon Redshift is a cloud data warehouse designed specifically for OLAP.
- Amazon Relational Database Service (Amazon RDS) is a relational database with OLAP functionality. You can use it to run OLTP workloads or with Oracle OLAP to perform complex queries on dimensional cubes.
- Amazon Aurora is a MySQL- and PostgreSQL-compatible cloud relational database that can run both OLTP and complex OLAP workloads.

Recommendations for Normalization between OLAP and OLTP systems

OLAP (Online Analytical Processing) and **OLTP (Online Transaction Processing)** systems differ due to their distinct purposes and usage patterns. Here's a breakdown:

1. Normalization in OLTP Systems:

OLTP systems focus on daily transactional data operations like inserting, updating, and deleting data quickly. Normalization in OLTP databases is critical to ensure data integrity, eliminate redundancy, and improve data efficiency.

Recommendations for OLTP:

- **High Normalization (3NF and above)**: OLTP databases should follow a highly normalized structure, often up to the Third Normal Form (3NF) or beyond. This helps reduce data redundancy, ensuring that each piece of information is stored only once. It makes updates efficient and maintains consistency across the system.
    - **1NF (First Normal Form)**: Ensure that the table has no repeating groups, and each field contains atomic values.
    - **2NF (Second Normal Form)**: All non-key attributes must depend on the primary key, eliminating partial dependency.
    - **3NF (Third Normal Form)**: Eliminate transitive dependencies, where non-key attributes depend on other non-key attributes.

The goal is to make the system efficient for fast transactional operations like insertions and updates while maintaining data consistency.

2. Normalization in OLAP Systems:

OLAP systems are designed for complex queries and reporting, where data is analyzed and aggregated over time. The focus is on read-heavy operations like running complex queries for reports and trends, rather than real-time updates or inserts.

Recommendations for OLAP:

- **Denormalization**: Unlike OLTP, OLAP systems often use denormalized structures. This means merging related tables and duplicating some data for faster querying and easier aggregation. In OLAP, data redundancy is acceptable because the focus is on optimizing read performance, not minimizing storage or maintaining quick updates.
    - **Star Schema**: This is a common design where a central fact table is surrounded by dimension tables. Each dimension is denormalized to allow quicker joins and easier reporting.
    - **Snowflake Schema**: A variation of the star schema, but more normalized. Dimension tables are further divided into additional related tables. This increases the complexity but reduces redundancy, offering a middle ground.

Denormalization helps OLAP systems avoid the need for multiple joins in complex queries, making analysis faster, especially with large datasets.

Key Differences:

Feature	OLTP Normalization	OLAP Denormalization
Purpose	Fast, frequent transactional operations	Complex queries, reporting, and analysis

Feature	OLTP Normalization	OLAP Denormalization
Normalization Level	High (up to 3NF or higher)	Low (denormalized, star or snowflake schema)
Data Redundancy	Minimized	Acceptable to improve query performance
Query Complexity	Simple queries involving small datasets	Complex queries involving large datasets
Update Frequency	Frequent updates and inserts	Infrequent bulk loading and queries
Join Operations	Efficient joins due to normalized structure	Avoids multiple joins by denormalizing data

Why These Differences Matter:

- **OLTP**: Normalization is key to ensure consistency and avoid data anomalies, especially when handling frequent updates. It also minimizes storage by eliminating redundant data.
- **OLAP**: Denormalization is used to optimize read-heavy queries where performance is prioritized. Since updates are less frequent, maintaining multiple copies of data is not a major concern.

In summary:

- **OLTP systems** use **highly normalized structures** for efficient transaction processing and data integrity.
- **OLAP systems** use **denormalized structures** to optimize for complex queries and reporting performance.

SQL Query Optimizations

Poorly written SQL queries can result in slow performance, high resource costs, locking and blocking issues, and unhappy users. The following are common practices that can be used to write efficient queries.

1. Use Indexes Wisely

Indexes let the database quickly look up rows instead of scanning the entire table. For Example:

Creating an index on customer_id if there are frequent queries on this column like the following query.

*SELECT * FROM orders WHERE customer_id = 123;*

CREATE INDEX idx_orders_customer_id ON orders(customer_id);

The above query will run much faster if **customer_id** is indexed.

- **Primary Index**: Automatically created on the primary key; ensures uniqueness and efficient access.
- **Secondary Index**: Created on non-primary key columns to improve query performance. Need to be created explicitly.
- **Clustered Index**: Determines the physical order of data in a table; only one per table. In some databases primary indexes are automatically clustered.
- **Non-Clustered Index**: Contains pointers to the data; multiple can exist per table

Best Practices:

- Index columns that are frequently used in WHERE clauses, JOIN conditions, or ORDER BY clauses.
- We should avoid over indexing as it makes insertions and deletions slower. Additional storage space required; can slow down write operations like INSERT, UPDATE, and DELETE due to index maintenance overhead.
- Regularly monitor and analyze index usage to optimize performance

2. Avoid SELECT *: Choose Only Required Columns

Using SELECT * impact query performance with large tables or joins. The database engine retrieves every column, even the ones you don't need which increases memory usage, slows down data transfer, and makes the execution plan more complex. **Example:**

Avoid this:

*SELECT * FROM products;*

Use this instead:

SELECT product_id, product_name, price FROM products;

Why This Helps:

- Reduces I/O load and memory usage.
- Enables the optimizer to skip unnecessary columns.
- Makes queries more readable and maintainable.

3. Limit Rows with WHERE and LIMIT

Fetching more rows than needed is a common issue. Even if you only use 10 rows in your app, the query might retrieve thousands, slowing things down. Use the WHERE clause to filter data precisely and LIMIT to restrict the number of rows returned.

Example:

SELECT name FROM customers

WHERE country = 'USA'

ORDER BY signup_date DESC

LIMIT 50;

Benefits:

- Reduces CPU and network load.
- Avoids returning excessive data during analysis or validation.
- Great for previewing transformations and testing queries.

4. Write Efficient WHERE Clauses

The WHERE clause is one of the most important parts of an SQL query because it filters rows based on conditions. However, how you write it can significantly impact performance. A common mistake is using functions or operations directly on column values in the WHERE clause - this can prevent the database from using indexes, which slows down query execution.

Poor:

*SELECT * FROM employees WHERE YEAR(joining_date) = 2022;*

Why this is bad: The YEAR() function is applied to every value in the joining_date column. This disables the use of indexes, forcing a full table scan.

Optimized:

*SELECT * FROM employees*

WHERE joining_date >= '2022-01-01' AND joining_date < '2023-01-01';

Optimization Tips:

- Don't use functions on columns (LOWER(column), YEAR(column), etc.)
- Avoid mathematical operations like salary + 5000 = 100000
- Rewrite conditions to let the database use available indexes

5. Avoid Functions on Indexed Columns

Using SQL functions (like UPPER(), LOWER(), DATE()) on indexed columns can prevent the database from using indexes, leading to slower queries.

Bad (pseudocode):

*SELECT * FROM users WHERE UPPER(email) = 'JOHN@GMAIL.COM';*

Good (pseudocode):

*SELECT * FROM users WHERE email = 'john@gmail.com';*

Why This Matters:

- Preserves index usability
- Keeps search efficient

- Prevents unnecessary computation on large datasets

6. Use Joins Smartly

Always join only the tables you need and filter data before joining whenever possible. Use INNER JOIN instead of OUTER JOIN when you don't need unmatched records.

Example:

SELECT u.name, o.amount
FROM users u
JOIN orders o ON u.user_id = o.user_id
WHERE o.amount > 100;

Why This Matters:

- Reduces join processing time
- Prevents Cartesian products (when no ON condition)
- Helps the planner optimize the execution path

7. Avoid N+1 Query Problems

N+1 happens when your app makes one query to get a list, then runs additional queries in a loop to get related data. Always aim to fetch related data in one query using JOINs.

Bad (pseudocode):

*SELECT * FROM users;*
*-- For each user: SELECT * FROM orders WHERE user_id = ?*

Good pseudocode:

SELECT u.user_id, u.name, o.order_id, o.amount
FROM users u
JOIN orders o ON u.user_id = o.user_id;

Why This Matters:

- Reduces database calls
- Improves latency and throughput
- Keeps the database from getting overloaded

8. Use EXISTS Instead of IN (for Subqueries)

When checking existence, EXISTS can be more efficient than IN, especially if the subquery returns a large dataset.

Bad (pseudocode):

SELECT name FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders);

Good (pseudocode):

SELECT name FROM customers
WHERE EXISTS (
SELECT 1 FROM orders WHERE orders.customer_id = customers.customer_id
);

Why This Matters:

- EXISTS stops searching once a match is found
- Often better optimized by query planners
- Reduces memory use for large subqueries

9. Avoid Wildcards at the Start of LIKE

Using % at the beginning of a LIKE pattern disables index use and leads to full table scans.

Bad (pseudocode):

*SELECT * FROM users WHERE name LIKE '%john';*

Good (pseudocode):

*SELECT * FROM users WHERE name LIKE 'john%';*

Why This Matters:

- Keeps searches index-friendly
- Speeds up pattern matching
- Reduces scan overhead

10. Consider Denormalization for Performance

While normalization keeps data clean, excessive joins can slow down read-heavy queries. Denormalization (storing some redundant data) can help in scenarios where performance is more critical than strict data structure.

Example:

- Store total order amount in the orders table instead of calculating with JOINs every time

Why This Matters:

- Reduces join complexity
- Speeds up frequent reads
- Useful for analytics or dashboards

11. Use Query Execution Plan

Every major DBMS has a way to show the execution plan (like EXPLAIN in MySQL/PostgreSQL). It shows how the SQL engine processes your query.

Example:

*EXPLAIN SELECT * FROM orders WHERE user_id = 42;*

Why This Matters:

- Helps identify full table scans
- Reveals if indexes are used
- Guides optimization decisions

12. Use UNION ALL Instead of UNION (if possible)

UNION removes duplicates, which adds sorting overhead. If you don't need duplicates removed, UNION ALL is faster.

Bad (pseudocode):

SELECT col FROM table1
UNION
SELECT col FROM table2;

Good (pseudocode):

SELECT col FROM table1
UNION ALL
SELECT col FROM table2;

Why This Matters:

- Avoids unnecessary sorting
- Faster result merging
- Better for performance on large sets

13. Avoid SELECT Inside Loops (In Applications)

Don't run a query inside a loop in your app if you can write one efficient query that retrieves all needed data.

Bad (pseudocode):

for id in ids:
cursor.execute("SELECT name FROM users WHERE id = ?", (id,))

Good (pseudocode):

SELECT id, name FROM users WHERE id IN (1, 2, 3, 4);

Why This Matters:

- Reduces round trips to the database
- Lower latency
- Better network and CPU usage

14. Partition Large Tables

Partitioning helps by breaking a large table into smaller, more manageable chunks. Queries on partitions are faster as they scan only relevant data.

Example:

- Partition sales data by year or region

Why This Matters:

- Speeds up scans on large datasets
- Helps with parallelism and archiving
- Makes indexes more effective within partitions

15. Optimize ORDER BY and GROUP BY

Sorting and grouping can be expensive operations. Always limit the number of rows being sorted or grouped and use indexes that match the ORDER BY columns if possible.

Bad (pseudocode):

*SELECT * FROM orders ORDER BY created_at;*

Good (pseudocode):

SELECT order_id, amount FROM orders WHERE created_at >= '2023-01-01' ORDER BY created_at;

Why This Matters:

- Reduces memory load for sorting
- Makes sorting and grouping faster
- Enhances user-facing query performance

Complex SQL queries, CTEs, window functions

Understanding complex SQL queries: CTEs and window functions

Complex SQL queries are vital for advanced data analysis and database management. They enable intricate data transformations at scale for tasks like data warehousing, ETL (Extract, Transform, Load), data analysis, and reporting.

1. Common Table Expressions (CTEs)

- What they are: CTEs are temporary named result sets defined within the scope of a single SQL statement (SELECT, INSERT, UPDATE, MERGE, or DELETE) using the WITH clause. They are not stored in the database but exist only during the query's execution.
- Purpose: Simplify complex queries and enhance code readability. Break down complex logic into manageable and reusable parts.
- Syntax:

sql

```
WITH cte_name (column1, column2, ...) AS (
  SELECT ...
```

```
 FROM ...
 WHERE ...
)
SELECT ...
FROM cte_name
WHERE ...;
```

Use code with caution.

- cte_name: Unique name for the CTE.
- column1, column2, ...: Optional list of column names for the CTE.
- SELECT ... FROM ... WHERE ...: The query that defines the CTE's result set.

- Benefits:
  - Improved Readability and Maintainability: Break down complex queries into smaller, easier-to-understand modules.
  - Reusability: Reference a single CTE multiple times within the same query, reducing code duplication.
  - Recursive Queries: Essential for working with hierarchical or iterative data structures, like organizational charts or bill of materials.
- Limitations:
  - Temporary Scope: Exist only for the duration of the query.
  - Potential Performance Issues: Can lead to performance degradation on very large datasets or with multiple references in some scenarios.
  - Not Allowed in All Database Operations: Restrictions may exist in certain operations like INSERT and UPDATE in some databases.

2. Window functions

- What they are: Functions that perform calculations across a defined set of rows ("window") related to the current row in the result set. Unlike aggregate functions, they don't collapse rows but return a value for each row.
- Syntax:

sql

```
function_name (column_name) OVER (
  [PARTITION BY column_name(s)]
  [ORDER BY column_name(s) [ASC | DESC]]
  [frame_clause]
```

)

Use code with caution.

  - function_name: The window function to use (e.g., ROW_NUMBER, SUM, RANK, LEAD, LAG).
  - column_name: The column on which to perform calculations.
  - PARTITION BY: Divides the result set into partitions (groups) for calculations.
  - ORDER BY: Specifies the order of rows within each partition.
  - window_frame: Defines the range of rows within each partition for the function to operate on (e.g., ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW).

- Key Concepts:
  - Partitions: Subsets of the result set defined by PARTITION BY, treated as separate groups for calculations.
  - Order: The sequence of rows within each partition defined by ORDER BY.
  - Window Frame: The specific subset of rows within a partition that the function operates on, defined by ROWS or RANGE.
- Types of Window Functions:
  - Ranking Functions: Assign a rank to rows (ROW_NUMBER, RANK, DENSE_RANK).
  - Aggregate Functions: Calculate aggregates over a window (SUM, AVG, MIN, MAX, COUNT).
  - Value Functions: Access values from other rows within the window (LEAD, LAG, FIRST_VALUE, LAST_VALUE).
- Practical Use Cases:
  - Running Totals: Calculating cumulative sums, for example.
  - Ranking: Ranking items or employees based on criteria within groups.
  - Moving Averages: Calculating averages over a sliding window.
  - Comparing Rows: Comparing current row values with preceding or succeeding row values.

3. Complex queries best practices

- Use Descriptive Naming: For CTEs, choose names that clarify their purpose.
- Refactor Complex Logic: Break down large queries into smaller, more manageable parts, potentially using CTEs or subqueries.
- Optimize JOINs and Subqueries: Prefer JOINs over subqueries when possible, especially correlated subqueries, for potentially faster execution.

- Use Indexes Effectively: Create or modify indexes on columns used in JOIN, WHERE, and ORDER BY clauses to improve performance.
- Avoid Functions on Indexed Columns: Applying functions to indexed columns in WHERE clauses can prevent indexes from being used effectively.
- Analyze Execution Plans: Tools like EXPLAIN or the SQL Server Execution Plan help visualize the query execution process, identifying bottlenecks like full table scans or inefficient joins.
- Limit Data Processing Early: Use WHERE clauses to filter data before performing JOINs or sorting to reduce the amount of data processed.

By effectively employing CTEs and window functions in conjunction with these best practices, you can write more efficient, readable, and powerful SQL queries for tackling complex data analysis and transformation tasks.

Window functions

What is a window function?

A **window function** performs a calculation across a set of table rows that are somehow related to the current row.
Unlike GROUP BY aggregates, window functions **do not collapse rows** - each input row remains in the result, and the window function returns an additional value computed from the window (the "frame") around that row.

Basic form:

<window_function>(...) OVER (

[PARTITION BY <expr_list>]

[ORDER BY <expr_list>]

[<frame_clause>]

)

- PARTITION BY divides the rows into groups (like GROUP BY but keeps rows).
- ORDER BY defines ordering inside each partition - required for ranking and cumulative functions.
- frame_clause further restricts which rows in the partition are part of the window for the current row (e.g., sliding window, running total).

Types of window functions (with short uses)

- **Aggregate functions used as window functions**: SUM(), AVG(), COUNT(), MAX(), MIN() - e.g., running totals, moving averages.
- **Ranking functions**: ROW_NUMBER(), RANK(), DENSE_RANK(), NTILE(n) - for top-N, ties, buckets.

- **Value functions**: LAG(), LEAD(), FIRST_VALUE(), LAST_VALUE() - access neighboring or extreme values.
- **Distribution functions**: PERCENT_RANK(), CUME_DIST() - for percentile-ish results.
- **Other**: DB-specific functions like NTH_VALUE() etc.

Example dataset (small, easy to follow)

Table sales:

id	seller	sale_date	amount
1	A	2025-08-01	100
2	B	2025-08-01	50
3	A	2025-08-02	200
4	A	2025-08-02	150
5	B	2025-08-03	300
6	C	2025-08-04	100

(We'll use this to compute examples.)

---

1) Running total (cumulative sum)

```
SELECT id, seller, sale_date, amount,
    SUM(amount) OVER (
     ORDER BY sale_date
     ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_total
FROM sales
ORDER BY sale_date, id;
```

Output (ordered sale_date, id):

id	sale_date	amount	running_total
1	2025-08-01	100	100
2	2025-08-01	50	150
3	2025-08-02	200	350
4	2025-08-02	150	500

id	sale_date	amount	running_total
5	2025-08-03	300	800
6	2025-08-04	100	900

Note: we used ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW so the running total is a physical cumulative sum up to the current row.

ROWS vs RANGE (important difference)

If you instead use:

SUM(amount) OVER (ORDER BY sale_date) -- default frame is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

RANGE treats *all rows with equal ORDER BY values as peers*. So both rows with sale_date = 2025-08-01 will see the sum of **both** 08-01 rows. With our data the RANGE-based running totals would be:

- For both 08-01 rows: 150 (100 + 50)
- For both 08-02 rows: 500 (100 + 50 + 200 + 150)
- etc.

Rule of thumb: use ROWS when you need a physical offset (e.g., last 7 rows), use RANGE when you want logical partitions by the ORDER BY expression (e.g., by date value).

---

2) Partitioned running total (per-group)

Running total per seller:

```
SUM(amount) OVER (
  PARTITION BY seller
  ORDER BY sale_date
  ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
) AS running_by_seller
```

For seller A (100, 200, 150) -> running totals: 100, 300, 450.
Seller B -> 50, 350. Seller C -> 100.

PARTITION BY is like a GROUP BY that doesn't collapse rows - it just resets the window per partition.

---

3) LAG / LEAD - compare current row to neighbour

Find change vs previous sale for each seller:

```
SELECT id, seller, sale_date, amount,
```

```
    LAG(amount)  OVER (PARTITION BY seller ORDER BY sale_date, id) AS prev_amount,
    amount - LAG(amount) OVER (PARTITION BY seller ORDER BY sale_date, id) AS delta
FROM sales
ORDER BY seller, sale_date, id;
```

For seller A:

- first sale: prev_amount = NULL
- second: prev_amount = 100, delta = 200 - 100 = 100
- third: prev_amount = 200, delta = 150 - 200 = -50

LAG and LEAD are great for diffs, percent changes, and "previous/next" comparisons.

---

4) Rankings: ROW_NUMBER, RANK, DENSE_RANK, NTILE

Assume we want a global ranking by amount DESC:

- ROW_NUMBER() - unique sequential number (breaks ties arbitrarily or by ORDER BY tie-breaker).
- RANK() - gaps after ties. If two rows tie at rank 4, next rank is 6.
- DENSE_RANK() - no gaps: ranks increase by 1 for each distinct value.
- NTILE(n) - bucket rows into n roughly equal groups.

Example ordering by amount (desc): amounts: 300, 200, 150, 100, 100, 50

id	amount	ROW_NUMBER	RANK	DENSE_RANK
5	300	1	1	1
3	200	2	2	2
4	150	3	3	3
1	100	4	4	4
6	100	5	4	4
2	50	6	6	5

Use ROW_NUMBER() inside a CTE/subquery to implement **top-N per group**:

```
WITH ranked AS (
  SELECT *,
      ROW_NUMBER() OVER (PARTITION BY seller ORDER BY amount DESC) AS rn
  FROM sales
```

)

SELECT * FROM ranked WHERE rn <= 3;

(That picks top 3 sales per seller.)

Side note: some SQL dialects (Snowflake, BigQuery) support QUALIFY to filter by window function directly: ... QUALIFY ROW_NUMBER() OVER (...) <= 3. Not all DBs have QUALIFY.

---

5) FIRST_VALUE / LAST_VALUE

FIRST_VALUE(x) and LAST_VALUE(x) return the first/last value *within the window frame*. **Important**: when you include ORDER BY but omit a frame clause, many DBs use RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW as the default frame. That means LAST_VALUE() will often return the **current row's value**, because the default frame ends at CURRENT ROW. If you want the actual last value of the partition, you must expand the frame:

Correct way to get the true partition last value:

LAST_VALUE(amount) OVER (

PARTITION BY seller

ORDER BY sale_date

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

) AS seller_last_amount

Alternatively, for "last by date" it's often simpler & faster to use MAX(amount) OVER (PARTITION BY seller) if you only need the maximum, or use a subquery to pick the last row.

---

6) Distribution: PERCENT_RANK and CUME_DIST

- PERCENT_RANK() = (rank - 1) / (rows_in_partition - 1) - returns 0..1 (0 for lowest rank).
- CUME_DIST() = (# of rows with value <= current row) / rows_in_partition - cumulative fraction.

Example for amounts [300,200,150,100,100,50] (N=6):

- For 300: PERCENT_RANK = (1-1)/(6-1) = 0.0, CUME_DIST = 1/6 ≈ 0.1667.
- For 100 (tie of two rows): PERCENT_RANK = (4-1)/(6-1) = 3/5 = 0.6, CUME_DIST = 3/6 = 0.5 (because three rows ≤ 100).

Useful for percentiles, scoring, and cumulative metrics.

---

7) Window functions with aggregates (combined patterns)

Compute per-customer total and share of the grand total:

WITH customer_totals AS (

```
  SELECT customer_id, SUM(amount) AS total
  FROM orders
  GROUP BY customer_id
)
SELECT customer_id,
    total,
    total / SUM(total) OVER () AS pct_of_grand_total,
    RANK() OVER (ORDER BY total DESC) AS total_rank
FROM customer_totals;
```

SUM(total) OVER() is the grand total across rows - very handy.

---

8) Where you can (and can't) use window functions

- You **cannot** use window functions in WHERE (because WHERE runs before windowing). Use a subquery/CTE or QUALIFY where supported.
- You **can** use window functions in SELECT, ORDER BY, and in an outer query's WHERE (after computing them in a subquery/CTE).
- Window functions can be used after GROUP BY - e.g., run window functions over grouped results.

---

9) Performance & best practices

- Window functions can be expensive, especially with large partitions or complex frames.
- Indexes that match PARTITION BY and ORDER BY may help (DB-dependent).
- Prefer computing on pre-aggregated data when possible (i.e. do GROUP BY in a CTE, then window functions on that smaller set).
- Avoid unnecessarily wide frames (e.g., UNBOUNDED FOLLOWING) if you only need recent rows.
- If you only need the partition max/min, MAX() OVER (PARTITION BY …) is usually faster than LAST_VALUE() with a big frame.
- Test on representative data sizes and check your DB's execution plan.

---

10) Common gotchas recap

- LAST_VALUE() default frame often returns current row → explicitly set the frame if you mean partition end.

- RANGE vs ROWS-duplicates in ORDER BY may cause surprising results with RANGE.
- Window functions do not reduce rows; to filter by them use subqueries/CTEs or QUALIFY.
- DISTINCT inside windowed aggregates is DB-specific - if you need distinct semantics, often do the distinct in a subquery.
- You cannot nest window functions directly (e.g., SUM(ROW_NUMBER() OVER (...))) - use subqueries.

Mini cheat sheet

- SUM(x) OVER (ORDER BY date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) - 7-row moving sum
- AVG(x) OVER (PARTITION BY category ORDER BY date ROWS BETWEEN 29 PRECEDING AND CURRENT ROW) - 30-day moving avg (when rows represent days)
- ROW_NUMBER() OVER (PARTITION BY dept ORDER BY salary DESC) - top-per-department
- LAG(value, 1) OVER (ORDER BY time) - previous value
- PERCENT_RANK() OVER (ORDER BY score DESC) - relative percentile-like rank

---

3. ETL / ELT & Data Pipeline Design

Difference between Batch Processing and Stream Processing

Today, an immense amount of data is generated, which needs to be managed properly for the efficient functioning of any business organization. Two clear ways of dealing with data are the batch and stream processes. Even though both methods are designed to handle data, there are significant differences in terms of working, application, and advantages. To make the right decision for optimizing the data flow, let's discuss the definitions of batch processing and stream processing.

What is Batch Processing?

Batch processing refers to the processing of a high volume of data in a batch within a specific time span. It processes large volumes of data all at once. Batch processing is used when data size is known and finite. It takes a little longer time to process data. It requires dedicated staff to handle issues. A batch processor processes data in multiple passes. When data is collected over time and similar data batched/grouped together then in that case batch processing is used.

Challenges With Batch processing

- Debugging these systems is difficult as it requires dedicated professionals to fix the error.
- Software and training require high expenses initially just to understand batch scheduling, triggering, notification, etc.

Advantages of Batch Processing

- **Efficiency in Handling Large Volumes:** Batch processing is very efficient when handling big volumes of data because it combines the data and process it at once.
- **Reduced Costs:** As the processing is in mass, it isn't very intensive and in some cases can be done outside the business hours and many a times saves the expenses.
- **Simplified Error Handling:** Batch processing errors are also easy to correct since the data is processed as a batch and an audit performed.

Disadvantages of Batch Processing

- **Delayed Results:** I want to note that this approach is good only for those tasks where the processing is done with a considerable time delay.
- **Inflexibility:** When a particular batch job is initiated, it becomes a bit difficult to introduce an alteration or provide means to process the new inputs until the current batch has been processed.

What is Stream Processing?

Stream processing refers to processing of continuous stream of data immediately as it is produced. It analyzes streaming data in real time. Stream processing is used when the data size is unknown and infinite and continuous. It takes few seconds or milliseconds to process data. In stream processing data output rate is as fast as data input rate. Stream processor processes

data in few passes. When data stream is continuous and requires immediate response then in that case stream processing is used.

Challenges with Stream processing

- Data input rate and output rate sometimes creates a problem.
- Cope with huge amount of data and immediate response.

Advantages of Stream Processing

- **Real-Time Processing:** Real time processing is made possible by stream processing, which outputs both results and actions.
- **Continuous Data Handling:** The real-time processing is ideal for the set-up where there is a constant stream of data that need to be analyzed as soon as possible.
- **Scalability:** The variability of data flooding can be managed in stream processing systems, thus making them effective for large scale data systems.

Disadvantages of Stream Processing

- **Complexity:** Stream processing systems, in its totality, is a complex area to implement and manage thus needs special skills.
- **Higher Costs:** Real-time processing requires more computer power thus be expensive as compared to batch processing.

Difference Between Batch Processing and Stream processing

The main differences between the two are:

- **Data Processing Approach**: Batch processing involves processing large volumes of data at once in batches or groups. The data is collected and processed offline, often on a schedule or at regular intervals. Stream processing, on the other hand, involves processing data in real-time as it is generated or ingested into the system. The data is processed as a continuous stream, with results generated in near real-time.
- **Data Latency**: Batch processing is typically slower than stream processing since the data is processed in batches, which can take some time. Stream processing, on the other hand, provides real-time results with low latency, making it suitable for applications that require immediate responses.
- **Data Volume**: Batch processing is suitable for processing large volumes of data, as it can be processed in batches, making it easier to manage and optimize. Stream processing, on the other hand, is designed to handle high volumes of data, which is processed in real-time.
- **Processing Complexity**: Batch processing is generally less complex than stream processing since the data is processed offline and in batches. Stream processing is more complex since it requires processing data in real-time, which can be challenging, especially for complex applications.
- **Processing Use Cases**: Batch processing is well-suited for use cases such as data warehousing, data mining, and data analytics, which involve processing large volumes

of historical data. Stream processing is suitable for use cases such as real-time monitoring, fraud detection, and IoT applications, which require real-time processing of data as it is generated.

Batch Processing	Stream Processing
Batch processing refers to processing of high volume of data in batch within a specific time span.	Stream processing refers to processing of continuous stream of data immediately as it is produced.
Batch processing processes large volume of data all at once.	Stream processing analyzes streaming data in real time.
In Batch processing data size is known and finite.	In Stream processing data size is unknown and infinite in advance.
In Batch processing the data is processes in multiple passes.	In stream processing generally data is processed in few passes.
Batch processor takes longer time to processes data.	Stream processor takes few seconds or milliseconds to process data.
In batch processing the input graph is static.	In stream processing the input graph is dynamic.
In this processing the data is analyzed on a snapshot.	In this processing the data is analyzed on continuous.
In batch processing the response is provided after job completion.	In stream processing the response is provided immediately.
Examples are distributed programming platforms like MapReduce, Spark, GraphX etc.	Examples are programming platforms like spark streaming and S4 (Simple Scalable Streaming System) etc.

Batch Processing	Stream Processing
Batch processing is used in payroll and billing system, food processing system etc.	Stream processing is used in stock market, e-commerce transactions, social media etc.
Processes data in batches or sets, typically stored in a database or file system.	Processes data in real-time, as it is generated or received from a source.
Processes data in discrete, finite batches or jobs.	Processes data continuously and incrementally.

workflow orchestration tools: what they are, how they work, examples, and the trade-offs between them.

---

1. What is workflow orchestration?

Workflow orchestration tools coordinate and manage the execution of tasks, jobs, or data pipelines — ensuring:

- **Order** (tasks run in the correct sequence)
- **Dependencies** (one task runs only after another succeeds/fails)
- **Scheduling** (run daily, hourly, event-based, etc.)
- **Monitoring & logging** (know when things fail)
- **Retries & error handling**
- **Scalability** (run across multiple machines/environments)

Think of it as the **conductor of an orchestra**, where each musician is a separate task — the conductor ensures they all play in the right order, at the right time, without chaos.

---

2. Why orchestration tools are needed

Without orchestration, you'd rely on:

- Bash scripts + cron jobs
- Manually triggered jobs
- Ad-hoc scripts in different places

That quickly leads to:

- Broken dependencies
- Hard-to-debug failures

- No centralized visibility
- Risk of overlapping or missed runs

An orchestration tool centralizes **control, visibility, and reliability**.

---

3. Core concepts in orchestration tools

Most orchestration tools share these concepts:

Concept	Meaning
Task	A single unit of work (e.g., run a Python script, SQL query, API call)
DAG (Directed Acyclic Graph)	A dependency graph where each node is a task and edges represent dependencies
Schedule	How often the workflow runs (cron-like, event-driven, manual)
Operators / Actions	Prebuilt task types (e.g., SQL query executor, file mover)
Sensors	Tasks that wait for a condition (e.g., file exists, API returns data)
Trigger rules	Whether a task runs on success, failure, or always
Execution context	Variables/configs passed to tasks for dynamic runs
Retry policy	Automatic reruns on failure
Monitoring/alerts	Logs, dashboards, notifications

---

4. Categories of workflow orchestration tools

A) General-purpose orchestrators

- **Apache Airflow** — Popular in data engineering; DAG-based; strong scheduling; Python DSL.
- **Prefect** — Python-first; modern UI; better for hybrid cloud/on-prem; less boilerplate than Airflow.
- **Luigi** — Python library for building pipelines; less UI-focused; older but still used.

B) Data & ML-specific

- **Dagster** — Data/ML pipelines with strong type checking; developer-friendly.
- **Metaflow** — From Netflix; focuses on ML workflows and reproducibility.
- **Kubeflow Pipelines** — ML pipelines running on Kubernetes.

C) Enterprise orchestration

- **Control-M** (BMC) — Legacy enterprise scheduling/orchestration.
- **Tivoli Workload Scheduler** (IBM) — Large enterprise batch jobs.

D) Cloud-native orchestration

- **AWS Step Functions** — Orchestrates AWS Lambda & services.
- **Google Cloud Workflows** — Orchestration for GCP services.
- **Azure Data Factory** — ETL orchestration with drag-and-drop UI.

E) Container-native orchestration

- **Argo Workflows** — Kubernetes-native, YAML-defined workflows.
- **Tekton Pipelines** — Kubernetes CI/CD pipelines.

---

5. Example — Airflow

Architecture

- **Scheduler**: Decides what tasks should run next.
- **Executor**: Runs tasks (can be local, Celery, Kubernetes, etc.).
- **Metadata DB**: Stores DAG states and logs.
- **Web UI**: Shows DAGs, runs, logs, retries.

Sample DAG in Airflow:

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime
def hello():
  print("Hello from task 1")

def world():
  print("Hello from task 2")

with DAG(
  dag_id='example_dag',
  start_date=datetime(2025, 8, 1),
  schedule_interval='@daily',
  catchup=False
```

```
) as dag:
    t1 = PythonOperator(task_id='say_hello', python_callable=hello)
    t2 = PythonOperator(task_id='say_world', python_callable=world)

    t1 >> t2  # t2 runs after t1
```

---

6. Example — AWS Step Functions

State machine defined in JSON/YAML:

```
{
  "Comment": "Example Step Function",
  "StartAt": "FirstTask",
  "States": {
    "FirstTask": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:REGION:ACCOUNT_ID:function:myFunction",
      "Next": "SecondTask"
    },
    "SecondTask": {
      "Type": "Pass",
      "End": true
    }
  }
}
```

- Integrates directly with AWS services.
- Can have retries, parallel branches, map loops.

---

7. Key differences between tools

Feature	Airflow	Prefect	Step Functions	Argo Workflows
Language	Python DAGs	Python	JSON/YAML	YAML
Deployment	Self-hosted/Cloud	Cloud/self-hosted	AWS only	Kubernetes

Feature	Airflow	Prefect	Step Functions	Argo Workflows
Scheduling	Yes	Yes	Event/time-based	Event/time
UI	Web UI	Modern UI	AWS Console	Argo UI
Retry logic	Yes	Yes	Yes	Yes
Event-driven	Limited	Good	Excellent	Good

---

8. How orchestration fits in the bigger picture

1. **Data ingestion** → Raw data lands in storage.
2. **Data processing tasks** → ETL jobs clean & transform it.
3. **Machine learning tasks** → Train & evaluate models.
4. **Deployment tasks** → Push results to dashboards, APIs, or alerts.
5. **Monitoring** → Detect & fix failures automatically.

Orchestration ensures all steps happen **in the right order, with reliability**.

---

9. Best practices

- **Idempotent tasks** — running twice should not break things.
- **Small, composable tasks** — easier retries & debugging.
- **Parameterize workflows** — avoid hardcoding dates/paths.
- **Centralized logging** — use the orchestrator's log aggregation.
- **Test locally** — run DAGs/workflows in dev before production.
- **Avoid over-orchestration** — not every micro-task needs to be a separate node; balance complexity.

Understanding Data Ingestion: Types, Techniques, and Tools

Discover the basics of data ingestion, techniques, tools, and best practices for managing different data types effectively!

Imagine you own an e-commerce company. To optimize your website and generate more sales, you can track customer behavior. This involves tracking every click, search query, and purchase, all of which provide valuable data.

However, the data you're collecting is scattered across multiple systems (e.g., web analytics platforms, transaction databases, customer support logs, etc.), which is a big problem.

To mitigate this problem, all the collected data should be consolidated into one place, and this is where data ingestion comes in handy.

Through data ingestion, you can pull all of this data into a central **data warehouse**, which enables you to gather real-time insights faster and perform analytics when necessary.

In this article, we will explore the fundamentals of data ingestion and the techniques used to facilitate the process. You'll learn about tools that help streamline data ingestion, common challenges that arise, and best practices to optimize the process for scalability, efficiency, and **data quality**.

Let's dive into it!

What is Data Ingestion?

Data ingestion is the process of collecting, importing, and loading data from various sources into a centralized system (e.g., a data warehouse, data lake, or cloud platform). This process enables you to consolidate data from different systems and make it available for further processing and analysis.

Efficient data ingestion is important because it forms the foundation for other data-driven operations, such as analytics, **visualization**, and sharing.

Types of Data Ingestion

Technically, data ingestion can be broadly categorized into **two primary types**:

- Batch ingestion
- Real-time ingestion

Each approach serves different needs and is suitable for various use cases depending on the speed and volume of data being processed.

Batch data ingestion

Batch ingestion refers to the process of collecting and processing data **at scheduled intervals**. For example, you may collect and process data hourly, daily, or even weekly. This approach is typically used when instantaneous insights are not required, and it helps manage large volumes of data efficiently.

Examples of batch ingestion use cases:

- **Reporting:** Collecting data periodically to generate business performance reports.
- **Periodic data syncs:** Synchronizing data across systems at regular intervals to keep them up-to-date.
- **End-of-day financial data analysis:** Aggregating transaction data to analyze financial performance after business hours.

Real-time data ingestion

Real-time (or streaming) data ingestion involves continuously collecting and processing data as it is generated. This method allows you to gain near-instantaneous insights, which is ideal for scenarios where time-sensitive decision-making is required.

Examples of real-time ingestion use cases:

- **Fraud detection:** Monitoring transactions in real time to identify and respond to suspicious activity.
- **IoT sensor data processing:** Gathering and processing data from connected devices, such as temperature or motion sensors, in real time.
- **Live analytics dashboards:** Providing up-to-the-minute data for dashboards that track key performance indicators (KPIs) and other metrics.

If you want to learn more about the difference between batching and streaming, scaling streaming systems, and real-world applications, I recommend the **Streaming Concepts** course.

Become a Data Engineer

Become a data engineer through advanced Python learning

Data Ingestion Techniques

Several techniques facilitate data ingestion. Each is designed to address different use cases and data processing needs.

The choice of technique you choose depends on 3 factors:

- The specific requirements of the business.
- The complexity of the data.
- The infrastructure in place.

In this section, I will touch on three data ingestion techniques and when they are most effective.

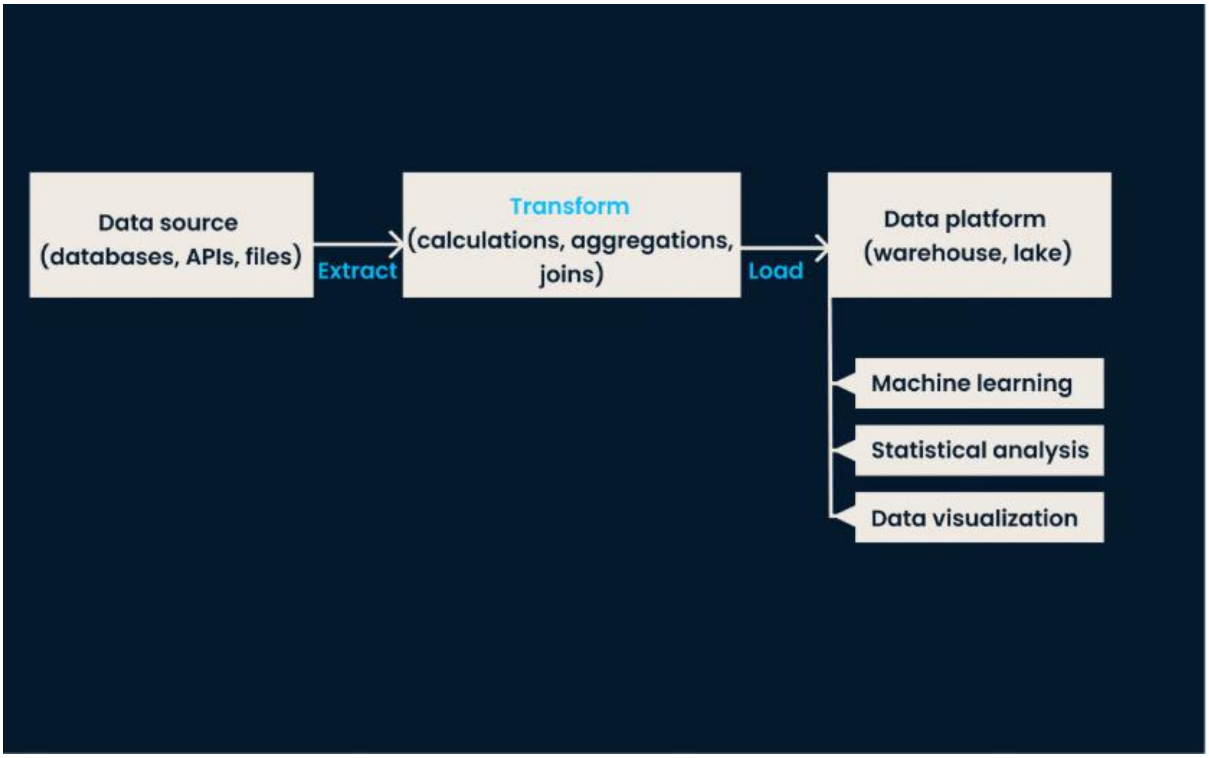
ETL (Extract, Transform, Load)

ETL is a traditional and widely used data ingestion technique. In this approach, data is **extracted** from various source systems, **transformed** into a consistent and clean format, and finally **loaded** into a data storage system, such as a data warehouse.

This method is highly effective for use cases that require consistent, structured data for reporting, analytics, and decision-making.

When to use ETL:

- When data needs to be **pre-processed** (cleansed or transformed) before it is loaded into the storage system.
- Scenarios where structured data is a priority, such as generating business reports or adhering to strict data quality requirements.
- When dealing with **relational databases** or smaller-scale data environments that prioritize clean, ready-to-analyze data.



ETL pipeline architecture.

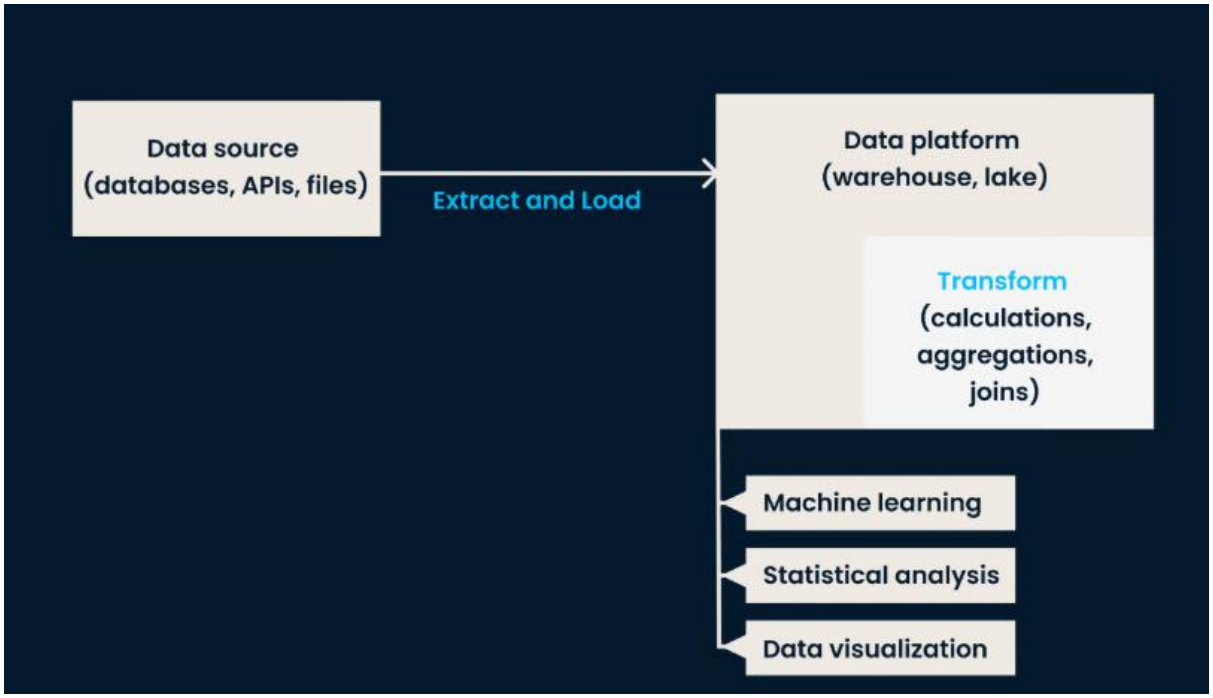
ELT (Extract, Load, Transform)

ELT reverses the order of operations compared to ETL. In this approach, data is **extracted** from source systems and **loaded** directly into the target system (usually a data lake or cloud-based platform). Once in the target storage system, the data is **transformed** as needed, often within the destination environment.

The ELT approach is particularly effective when dealing with large volumes of unstructured or semi-structured data, where transformation can be deferred until analysis is required.

When to use ELT:

- When working with data lakes or cloud storage, which can handle raw data at scale.
- When dealing with diverse, unstructured, or semi-structured data that may not need immediate transformation.
- When flexibility is needed, as ELT allows data transformations to be performed on demand within the storage environment.



ELT pipeline architecture.

Change data capture (CDC)

CDC is a technique for identifying and capturing changes (e.g., inserts, updates, deletes) made to data in source systems and then replicating those changes in real time to the destination systems.

This technique ensures that target systems are always up-to-date without reloading the entire dataset. CDC is highly effective for maintaining real-time synchronization between databases and ensuring consistency across systems.

When to use CDC:

- When real-time data synchronization is critical, such as in high-frequency trading, e-commerce transactions, or operational reporting.
- When you need to keep destination systems updated with minimal lag or data refresh times.
- When it is necessary to avoid the overhead of re-ingesting entire datasets.

Data Sources for Ingestion

We've established that data ingestion involves pulling information from various sources. But we haven't discussed what those structures are — and that's what we'll do in this section.

The data sources for ingestion generally fit into 3 main types:

- Structured
- Semi-structured
- Unstructured data

As you will learn when we examine the different sources, each requires different handling and processing techniques to be utilized effectively.

Structured data sources

Structured data is highly organized and follows a predefined schema, making it easier to process and analyze. It is typically stored in relational databases, which store and provide access to related data points. The data is arranged into tables with clearly defined rows and columns.

Due to this organized nature, structured data is the most straightforward to ingest.

Traditional tools like **MySQL** and **PostgreSQL** are typically used to handle structured data. These SQL-based tools simplify the validation process during ingestion by enforcing strict schema rules. When data is ingested into these systems, they automatically ensure it adheres to the structure defined by the database tables.

For example, if a column is set to accept only integer values, MySQL or PostgreSQL will reject any data that doesn't meet this requirement.

Other things to note about structured data are:

- It is typically collected in large volumes (and in a predictable format).
- It is often ingested through batch processes, which enable periodic updates to refresh the data as needed.

Semi-structured data sources

Semi-structured data, while still organized to some degree, does not conform to a rigid, predefined schema like structured data. Instead, it uses tags or markers to provide some organization, but the structure remains flexible.

This data type is typically found in formats like **JSON** files, CSV files, and **NoSQL databases**, and it is commonly used in web applications, APIs, and log systems.

Since the structure can vary, semi-structured data offers flexibility in storage and processing, making it suitable for systems that handle evolving or inconsistent data formats.

Handling semi-structured data requires tools to parse and extract meaningful information from the embedded tags or markers. For instance, a JSON file may contain a collection of key-value pairs, and the ingestion process would involve parsing these pairs and transforming them into a suitable format.

Semi-structured data is often ingested into flexible environments like data lakes or cloud platforms, where it can be stored in its raw form and processed later as needed.

Other things to note about semi-structured data include:

- Depending on the use case, it can be ingested in real-time or through scheduled batches.
- It requires transformation during ingestion to ensure it can be processed or analyzed effectively.

Unstructured data sources

Unstructured data is the most complex type to manage because it lacks any specific format or organizational structure. This category includes data like:

- Images
- Videos
- Audio
- Documents

Unlike structured or semi-structured data, unstructured data does not have clear rows, columns, or tags to define its content, making it more challenging to process and analyze.

Despite its complexity, unstructured data can hold valuable insights, particularly when analyzed using advanced technologies like **machine learning** or **natural language processing**.

Because unstructured data is not organized in a predefined way, specialized tools are required to handle the ingestion process. These tools can extract relevant information from raw, unstructured sources and transform it into formats that can be analyzed.

Lastly, unstructured data is typically stored in data lakes, where it can be processed as needed.

Comparison Table: Structured vs. Semi-Structured vs. Unstructured Data

Feature	Structured data	Semi-structured data	Unstructured data
Definition	Highly organized data with a predefined schema (e.g., rows and columns).	Partially organized data with tags or markers for flexibility (e.g., JSON, CSV).	Data without any predefined structure or organization (e.g., images, videos).
Examples	Relational databases like MySQL, PostgreSQL.	JSON files, NoSQL databases, CSV logs.	Images, videos, audio files, text documents.
Ease of processing	Easiest to process due to its strict schema.	Moderate difficulty; requires parsing and transformation.	Most challenging due to lack of structure.
Storage	Relational databases or data warehouses.	Data lakes, NoSQL databases, cloud storage.	Data lakes or specialized storage for raw files.

Ingestion approach	Typically ingested in batches.	Flexible; can be ingested in real-time or batches.	Primarily ingested in raw format for later processing.
Tools used	SQL-based tools like MySQL, PostgreSQL.	Tools that handle flexible schemas, such as MongoDB, Google BigQuery.	Advanced tools like Hadoop, Apache Spark, or AI/ML-based processing tools.
Use case examples	Generating reports, transactional systems.	API responses, web application logs.	Image recognition, speech analysis, sentiment analysis.

Challenges in Data Ingestion

Data ingestion comes with several challenges that you must address to ensure efficient and effective data management. In this section, we will discuss what these challenges are and solutions that can be employed to overcome them.

Handling large volumes of data

One of the primary challenges in data ingestion is the sheer volume of data that needs to be processed. As businesses grow, the amount of data they generate and collect can increase exponentially, particularly with the rise of real-time data streams from sources like IoT devices and social media platforms.

Scaling data ingestion processes to handle these growing volumes can be difficult, especially with streaming data that requires continuous ingestion.

Teams must ensure their infrastructure can support batch and real-time data ingestion at scale without degradation. The **Streaming Concepts** course is a great way to get started.

Data quality and consistency

Data quality and consistency are another significant challenge, especially when data is ingested from multiple disparate sources. Each source may have different formats, levels of completeness, or even conflicting information.

Ingesting data without addressing these inconsistencies can lead to inaccurate or incomplete datasets, eventually impacting downstream analytics.

Data validation checks must be implemented during ingestion to ensure that only high-quality data enters the system. Additionally, teams need to maintain consistency across various sources to avoid data discrepancies.

I recommend deep diving into this topic with the excellent Introduction to **Data Quality** course.

Security and compliance

When sensitive data is ingested, security and compliance become top priorities. Teams that handle personal information (e.g., customer names, addresses, payment details, etc.) must protect this data throughout the ingestion process.

For example, a financial services company must adhere to stringent regulations like GDPR or HIPAA when ingesting client data to avoid legal repercussions. Failure to secure data during ingestion could result in data breaches, compromising customer trust and leading to substantial financial penalties.

To address this, teams should implement encryption and secure access controls during the ingestion process. This ensures that sensitive data is protected from unauthorized access. Additionally, data must be ingested in a manner that complies with all relevant regulatory standards, with regular audits to verify ongoing compliance.

Discover how to keep data safe and secure with the beginner-friendly **Data Security** interactive course.

Latency requirements

In scenarios where immediate insights are needed, delays in data ingestion can result in outdated or irrelevant information. Low-latency data ingestion requires the ability to process and make data available for analysis almost instantaneously, which can be challenging when dealing with large volumes of data.

Teams must carefully optimize their systems to reduce latency and ensure data is available when needed.

Best Practices for Data Ingestion

There are a few best practices to ensure efficient and effective data ingestion.

Prioritize data quality

Ingested data will be used for analytics, reporting, and decision-making, so it must meet predefined standards of accuracy and completeness.

Implementing data validation checks during ingestion can help verify that data conforms to the expected formats and quality standards. I've written about this before in a **Great Expectations tutorial**.

For example, if data is expected to be in a certain range or meet specific criteria, use validation rules to automatically reject or flag any data that doesn't meet these standards. This minimizes errors and ensures that only reliable data is processed downstream.

Choose the right ingestion approach

Another best practice is to ensure you select the appropriate ingestion approach for your business needs.

Batch ingestion is typically used for less time-sensitive data, such as historical or periodic reports. In contrast, real-time ingestion is used for applications that require instant access to the latest data, such as fraud detection or IoT monitoring.

Assess your needs based on the frequency and urgency of data updates.

Ensure scalability

Your data volume will grow over time. Thus, it's important to select scalable tools and infrastructure to handle increasing amounts of data without compromising performance. This lets you expand your data pipelines and processing capabilities to accommodate future growth.

Use data compression

Less storage space required means lower infrastructure costs. Compression also speeds up data transfer, which means ingestion will be faster – mainly when dealing with large datasets or high-volume streams. This practice is especially beneficial when dealing with semi-structured or unstructured data, as they often come in large sizes.

Data Ingestion Tools

As you can see, data ingestion is a simple concept but it can be quite difficult to implement on your own. What are the best tools to help you with data ingestion? I've curated a list of the most commonly used.

Apache Kafka

Apache Kafka is a distributed streaming platform that excels at real-time data ingestion.

- It is designed to handle high-throughput, low-latency data streams, which makes it ideal for use cases that require instant access to large volumes of continuously generated data.
- A significant benefit of Kafka is that it enables businesses to process, store, and forward streaming data to various systems in real time.
- It is widely used in scenarios like fraud detection, IoT data processing, and real-time analytics.

The **Introduction to Kafka** course can teach you how to create, manage, and troubleshoot Kafka for real-world data streaming challenges.

Apache Nifi

Apache Nifi is a data ingestion tool known for its user-friendly visual interface.

- Its accessible interface enables businesses to design, monitor, and manage data flows using a drag-and-drop mechanism.
- Nifi supports many data sources—including batch and real-time data—and can integrate with systems such as Hadoop, relational databases, and NoSQL stores.
- Its ability to automate data ingestion workflows and handle complex data routing and transformation tasks makes it a flexible and powerful tool for small-scale and enterprise-level data ingestion needs.

Amazon Kinesis

Amazon Kinesis is a cloud-native service from AWS that specializes in ingesting, processing, and analyzing streaming data. It is designed for applications that require near real-time processing of high-throughput data streams, such as:

- Video feeds
- Clickstream data
- IoT sensor data

One of the benefits of Kinesis is its ability to **integrate with other AWS services** for storage, analytics, and machine learning.

This tool is particularly well-suited for organizations that leverage cloud infrastructure and seek a fully managed solution to handle their data ingestion requirements without concerning themselves with hardware or scalability issues.

Google Cloud Dataflow

Google Cloud Dataflow is a fully managed service provided by Google Cloud that supports both batch and real-time data processing.

- It is built on Apache Beam, an open-source unified programming model, and allows users to design complex data pipelines that can handle both large-scale batch processing and real-time streaming data.
- Dataflow automates resource management and scaling, making it ideal for businesses that want to process and analyze data without manual infrastructure management.

Airbyte

Airbyte is an open-source data ingestion platform that simplifies data integration from various sources into storage solutions.

- It offers pre-built connectors for hundreds of data sources and destinations and supports batch and real-time ingestion.
- The cool thing about Airbyte is that it is highly customizable—this allows users to extend connectors to fit their needs.
- Additionally, it provides a managed cloud service for those seeking a hands-off solution.

Data ingestion tools comparison

Tool	Type	Best for	Key features	Deployment	Ease of use	Customization
Apache Kafka	Streaming Platform	Real-time data ingestion	High-throughput, low-latency, scalable	Cloud & On-Prem	Mod erate	High

Tool	Type	Best for	Key features	Deployment	Ease of use	Customization
Apache Nifi	Data Flow Management	Automated data pipelines	Drag-and-drop UI, real-time processing	Cloud & On-Prem	High	High
Amazon Kinesis	Cloud Streaming Service	AWS-native real-time analytics	Fully managed, scalable, AWS-integrated	Cloud (AWS)	High	Moderate
Google Cloud Dataflow	Data Processing Service	Batch & streaming ETL	Auto-scaling, Apache Beam-based	Cloud (GCP)	High	Moderate
Airbyte	Data Integration Platform	Pre-built & custom connectors	Open-source, real-time & batch support	Cloud & On-Prem	High	Very High

Conclusion

In this article, we've explored the fundamentals of data ingestion, its types, and the techniques that enable it. You've learned about batch and real-time ingestion and popular approaches like ETL, ELT, and Change Data Capture (CDC). We've also covered best practices, the various data sources, and the challenges organizations face while performing.

For more insights into data management and ingestion tools, I recommend exploring the following resources:

- **ETL and ELT in Python**
- **Understanding Data Engineering**
- **Streaming Concepts**

data ingestion best practices so you can design ingestion pipelines that are **reliable, scalable, and future-proof**.

---

1. Understand What "Data Ingestion" Means

Data ingestion is the process of bringing data from one or more sources into a storage system (data lake, warehouse, operational DB, etc.) for further processing and analysis.

It can be:

- **Batch** — ingest chunks of data at intervals (e.g., every hour).
- **Streaming** — ingest continuously in near real-time.
- **Hybrid** — mix of both (common in modern architectures).

---

2. Best Practices Overview

I'll break them down into **planning, design, processing, and operations**.

A. Planning & Architecture

1. **Define the data contract early**
    - Agree on schema, formats, delivery frequency, and quality expectations with the data provider.
    - Avoid uncontrolled schema drift.
2. **Choose the right ingestion mode**
    - **Batch**: cheaper, simpler, good for historical data loads.
    - **Streaming**: needed for low-latency analytics, monitoring, or event-driven use cases.
    - Some tools (Kafka, Debezium, Flink) can do both.
3. **Design for scalability from the start**
    - Use horizontally scalable ingestion systems (e.g., Kafka, Kinesis, Pub/Sub, NiFi, Airbyte).
    - Partition data to allow parallel processing.
4. **Plan for incremental ingestion**
    - Avoid full reloads — use change data capture (CDC) or last-updated timestamps.

B. Data Handling & Quality

5. **Validate at the door**
    - Apply schema validation, type checks, and mandatory field checks at ingestion time.
    - Reject or quarantine bad data before it contaminates downstream systems.
6. **Preserve raw data**
    - Store an immutable raw copy in a data lake (Parquet/Avro/JSON) before transformations.
    - Allows reprocessing if transformation logic changes.
7. **Enforce consistent formats**
    - Prefer self-describing formats like Parquet/Avro for structured data.
    - Use standardized encoding (UTF-8) and date/time formats (ISO 8601).
8. **Track metadata**
    - Capture ingestion timestamps, source system identifiers, and processing job IDs.
    - This is crucial for lineage and debugging.

C. Performance & Reliability

9. **Implement backpressure handling**
    - In streaming, ensure your pipeline can slow down gracefully if downstream systems lag.
    - Tools like Kafka naturally buffer data.
10. **Enable retries & dead-letter queues**
    - Retries for transient failures.
    - Dead-letter queues (DLQs) for unprocessable events with logging for later inspection.
11. **Compress during transit**
    - Use gzip, Snappy, or ZSTD to reduce transfer costs and improve throughput.
12. **Secure the data pipeline**
    - Use TLS for data in transit.
    - Encrypt sensitive data at rest.
    - Use role-based access control (RBAC) for ingestion tools.

D. Monitoring & Operations

13. **Set up real-time monitoring**
    - Monitor ingestion latency, error rate, throughput.
    - Set alerts for spikes in failures or lag.
14. **Track data freshness**
    - Measure the gap between source data creation and ingestion completion.
    - Helps detect delays in upstream systems.
15. **Version control ingestion logic**
    - Store pipeline configs and code in Git.
    - Tag deployments to correlate issues with code changes.
16. **Test with production-like loads**
    - Catch bottlenecks early by simulating high-volume ingestion before go-live.

---

3. Tooling Examples

- **Batch ingestion**: Apache NiFi, AWS Glue, Azure Data Factory, Airbyte, dbt (for ELT).
- **Streaming ingestion**: Apache Kafka, Apache Flink, AWS Kinesis, Google Pub/Sub.
- **CDC ingestion**: Debezium, Fivetran, Striim.

---

4. Quick Diagram — Best Practice Flow

[Source Systems]

↓

[Schema Validation + Metadata Capture]

↓

[Raw Zone Storage (Immutable Copy)]

↓

[Transform/Clean (Optional in ELT)]

↓

[Data Warehouse / Analytics Layer]

↓

[Monitoring + Alerts + DLQ]

---

5. Key Takeaways

- Always **keep the raw** — it's your audit trail.
- Validate early to avoid downstream cleanup nightmares.
- Monitor continuously — ingestion failures silently kill dashboards.
- Design for **scalability and resilience** from day one.
- Secure both **data in transit** and **data at rest**.

Change Data Capture (CDC)

Change Data Capture (CDC) is a method used in databases to track and record changes made to data. It captures modifications like inserts, updates, and deletes, and stores them for analysis or replication. CDC helps maintain data consistency across different systems by keeping track of alterations in real-time. It's like having a digital detective that monitors changes in a database and keeps a log of what happened and when.

What is Change Data Capture (CDC) in System Design?

Change Data Capture (CDC) is an important component in system design, particularly in scenarios where real-time data synchronization, auditing, and analytics are crucial. CDC allows systems to track and capture changes made to data in databases, enabling seamless integration and replication across various systems.

- In system design, CDC facilitates the creation of architectures that support efficient data propagation, ensuring that updates, inserts, and deletes are accurately mirrored across different components or databases in real-time or near real-time.
- By incorporating CDC into system design, developers can enhance data consistency, improve performance, and enable advanced functionalities like real-time analytics and reporting.

Importance of Change Data Capture (CDC)

Change Data Capture (CDC) holds immense importance in facilitating real-time data synchronization and powering event-driven architectures.

1. **Real-time Data Synchronization:** CDC captures and propagates data changes as they occur, ensuring that all connected systems remain updated in real-time. This is crucial for scenarios where multiple systems or databases need to stay synchronized without delays, enabling seamless data sharing and consistency across the ecosystem.
2. **Event-Driven Architectures:** CDC serves as a cornerstone for event-driven architectures, where actions are triggered by events or changes in the system. By capturing data changes as events, CDC enables systems to react dynamically to these changes, initiating relevant processes or workflows in real time. This results in more responsive and agile systems that can adapt to changing conditions or requirements instantly.
3. **Efficient Data Processing:** CDC minimizes the need for manual intervention or batch processing by continuously streaming data changes. This leads to more efficient data

processing pipelines, reducing latency and ensuring that downstream systems have access to the latest information without waiting for scheduled updates.

4. **Scalability and Flexibility:** With CDC, event-driven architectures can scale easily to handle increasing data volumes and accommodate evolving business needs. By decoupling components and leveraging asynchronous communication, CDC enables systems to scale horizontally while maintaining responsiveness and reliability.
5. **Enhanced Analytics and Insights:** Real-time data synchronization facilitated by CDC enables organizations to derive insights from up-to-date data, driving informed decision-making and enabling timely actions. By integrating CDC with analytics platforms, organizations can gain immediate visibility into trends, patterns, and anomalies, empowering them to respond swiftly to changing market conditions or customer behaviors.

Change Data Capture (CDC) Principles

Below are the principles of Change Data Capture (CDC):

- **Capture:** CDC captures changes made to data in a source system, including inserts, updates, and deletes, without affecting the source's performance.
- **Log-based Tracking:** It leverages database transaction logs or replication logs to identify and extract data changes, ensuring accurate and reliable capture.
- **Incremental Updates:** Instead of transferring entire datasets, CDC focuses on transmitting only the changed data, minimizing network bandwidth and processing overhead.
- **Real-time or Near Real-time:** CDC operates in real-time or near real-time, ensuring that data changes are propagated to target systems promptly, maintaining data freshness.
- **Idempotent Processing:** CDC processes changes in an idempotent manner, ensuring that duplicate changes do not result in unintended side effects or data inconsistencies.

Use Cases of Change Data Capture (CDC)

Below are the use cases of Change Data Capture (CDC):

- **Data Warehousing:** CDC is used to replicate data from transactional databases to data warehouses, ensuring that analytical systems have access to the latest operational data for reporting and analysis.
- **Replication:** CDC facilitates database replication across geographically distributed environments, enabling disaster recovery, data distribution, and load balancing.
- **Data Integration:** CDC enables seamless data integration between heterogeneous systems, supporting scenarios such as integrating legacy systems with modern applications or synchronizing data between cloud and on-premises environments.
- **Real-time Analytics:** CDC powers real-time analytics platforms by continuously feeding data changes into analytical systems, enabling organizations to derive insights from fresh data and respond swiftly to changing conditions.

- **Data Synchronization:** CDC ensures data consistency across multiple systems by synchronizing data changes in real-time, supporting scenarios such as synchronization between operational databases and caching layers or between microservices in distributed architectures.

Applications of Change Data Capture (CDC)

Below are the applications of Change Data Capture (CDC):

- **Financial Services:** CDC is used in financial services for real-time fraud detection, risk management, and compliance monitoring by capturing and analyzing transactional data changes in real time.
- **E-commerce:** In e-commerce, CDC enables real-time inventory management, order processing, and personalized marketing by synchronizing data changes across multiple systems, such as inventory databases, order management systems, and customer relationship management (CRM) platforms.
- **Healthcare:** CDC is employed in healthcare for real-time patient monitoring, clinical decision support, and health information exchange by capturing and processing data changes from electronic health records (EHRs), medical devices, and healthcare information systems.
- **Logistics and Supply Chain:** CDC facilitates real-time tracking and optimization of logistics and supply chain operations by capturing and analyzing data changes from sensors, RFID tags, and inventory management systems, enabling efficient inventory management, route optimization, and supply chain visibility.
- **Telecommunications:** In telecommunications, CDC supports real-time billing, network optimization, and customer experience management by capturing and processing data changes from network elements, billing systems, and customer interaction channels, enabling operators to offer personalized services and ensure network reliability.

Change Data Capture (CDC) Implementation Patterns

CDC implementation patterns encompass various approaches and strategies for capturing, processing, and propagating data changes in real-time or near real-time. Here are some common CDC implementation patterns:

- **Log-based CDC:**
    - This pattern leverages database transaction logs or replication logs to capture data changes.
    - It involves monitoring and parsing database logs to extract change events, which are then propagated to target systems. Log-based CDC offers low latency and high accuracy, making it suitable for real-time data synchronization.
- **Trigger-based CDC:**
    - In this pattern, triggers are added to database tables to capture data changes as they occur. When an insert, update, or delete operation is performed on a table, the trigger executes custom logic to record the change event, which is then processed and propagated to target systems.

    - Trigger-based CDC is often used in scenarios where database logs are not accessible or reliable.

- **Change Data Publisher-Subscriber Model:**
    - This pattern involves a publisher-subscriber architecture, where data changes are published by the source system and subscribed to by one or more target systems.
    - The publisher captures data changes and publishes them to a message broker or event bus, while subscribers consume the change events and apply them to their respective databases or systems.
    - This decoupled approach enables scalability and flexibility in handling data changes across distributed environments.
- **Change Data Mesh:**
    - The Change Data Mesh pattern decentralizes CDC by distributing responsibility for capturing, processing, and consuming change events to individual services or domains within an organization.
    - Each service or domain is responsible for managing its own change data, allowing for greater autonomy and scalability in handling data changes.
    - Change Data Mesh promotes a decentralized, event-driven architecture that fosters agility and innovation.

Techniques for integrating CDC into existing data pipelines

Integrating Change Data Capture (CDC) into existing data pipelines requires careful planning and consideration of various techniques to ensure seamless data synchronization and processing. Here are several techniques for integrating CDC into existing data pipelines:

- **Change Data Capture Tools:** Utilize CDC tools and platforms specifically designed for integrating with existing data pipelines. These tools often provide out-of-the-box connectors and adapters for popular databases and messaging systems, simplifying the integration process. Examples include Debezium, Attunity, and Oracle GoldenGate.
- **Database Triggers:** Implement database triggers to capture data changes at the source. Triggers can be configured to execute custom logic whenever insert, update, or delete operations are performed on specific tables. This technique is particularly useful when direct access to database logs is not feasible or supported.
- **Log-based CDC:** Leverage log-based CDC techniques to capture data changes from database transaction logs or replication logs. Log-based CDC offers low latency and high fidelity by directly monitoring changes at the database level. Implement CDC solutions or frameworks like Apache Kafka Connect with Debezium, which can stream database change events from transaction logs into Kafka topics.
- **Message Queues and Event Streams:** Integrate CDC with message queues or event streams to decouple data producers from consumers in the pipeline. Use message brokers like Apache Kafka or cloud-based event streaming platforms such as Amazon

Kinesis or Google Cloud Pub/Sub to capture, buffer, and distribute change events to downstream systems.

- **Stream Processing:** Apply stream processing techniques to transform and enrich change data streams in real-time. Use frameworks like Apache Kafka Streams, Apache Flink, or Apache Spark Streaming to perform data processing tasks such as filtering, aggregating, and joining change events before they are consumed by downstream applications.
- **Error Handling and Retry Mechanisms:** Design robust error handling and retry mechanisms to handle failures and transient issues in the data pipeline. Implement strategies such as dead-letter queues, exponential backoff, and circuit breakers to manage exceptions and retries gracefully, ensuring fault tolerance and data integrity.

Best Practices for Scaling Change Data Capture (CDC) Solutions

Scaling Change Data Capture (CDC) solutions to handle large volumes of data changes requires a strategic approach to ensure performance, reliability, and efficiency. Here are some best practices to achieve this:

1. **Optimize Log-Based CDC:** For log-based CDC, ensure that the transaction logs are properly configured to retain necessary change data long enough for CDC processes to capture it. Use tools like Apache Kafka with Debezium, which are designed to handle high-throughput change streams efficiently.
2. **Partitioning:** Use data partitioning to distribute the workload across multiple nodes or instances. For example, partition Kafka topics based on logical keys (e.g., user ID, region) to ensure even distribution of change events and parallel processing.
3. **Batch Processing:** Where real-time processing is not critical, consider batching changes to reduce the overhead associated with processing each change individually. This can be done by configuring CDC tools to group changes into batches and process them periodically.
4. **Horizontal Scaling:** Design the CDC solution to scale horizontally by adding more instances or nodes to the system. Ensure that the CDC architecture supports distributed processing and load balancing.
5. **Efficient Storage:** Use high-performance, scalable storage solutions for capturing and storing change data. Cloud-based storage options like Amazon S3, Google Cloud Storage, or Azure Blob Storage can provide scalable and durable storage for CDC logs and snapshots.
6. **Load Balancing:** Distribute the CDC workload across multiple consumers or processors to avoid bottlenecks. Use load balancers or distributed stream processing frameworks to manage and balance the load effectively.

Ensuring consistency and reliability in Change Data Capture (CDC)

Ensuring consistency and reliability in Change Data Capture (CDC) systems is crucial for maintaining data integrity and trust in data synchronization processes. Here are several best practices to achieve this:

1. **Transactional Consistency:** Ensure that CDC captures changes within the context of database transactions. This means changes should only be captured once the transaction is committed, avoiding partial or incomplete data capture. Log-based CDC techniques typically support this by monitoring transaction logs.
2. **Idempotent Processing:** Design the CDC system to handle duplicate events gracefully. Each change event should be processed in an idempotent manner, meaning applying the same change multiple times will not affect the final result. This prevents data inconsistencies due to event duplication.
3. **Checkpointing and State Management:** Implement checkpointing to track the last successfully processed change. This allows the CDC system to resume from the last known good state after a failure, ensuring no data loss or duplication. Tools like Apache Kafka support offset management for this purpose.
4. **Schema Evolution Handling:** Manage schema changes to ensure that updates to the database schema do not break the CDC pipeline. Use schema registry tools to track and manage schema versions. Ensure the CDC system can handle backward-compatible schema changes gracefully.
5. **Data Validation and Consistency Checks:** Implement data validation mechanisms to verify the integrity and consistency of captured changes. This can include checksums, version numbers, or validation queries to compare source and target data periodically.
6. **Reliable Messaging:** Use reliable messaging systems to transport change events. Systems like Apache Kafka, RabbitMQ, or AWS Kinesis offer durability, fault tolerance, and guarantees on message delivery, ensuring that no changes are lost in transit.
7. **Version Control:** Use version control for CDC configurations and schemas. This allows for tracking changes and rolling back to previous versions if issues are detected. It also ensures that all components of the CDC system are synchronized and consistent.

Real-world Examples

Here are some real-world examples of successful Change Data Capture (CDC) implementations across different industries:

1. Netflix

Real-time data synchronization and analytics. Netflix uses a combination of Apache Kafka and Apache Flink for their CDC pipeline. Kafka captures changes from various data sources and streams them to Flink for real-time processing and analytics.

- This architecture supports various use cases such as monitoring streaming service usage, content recommendations, and fraud detection.
- Enhanced real-time data processing capabilities, improved user experience through personalized content, and efficient monitoring of streaming services.

2. Uber

Real-time data synchronization across multiple microservices and data stores. Uber employs Apache Kafka and their own open-source project, Cadence, for CDC. They use Kafka to capture changes from their transactional databases and propagate them to other systems in real time.

- Cadence helps in orchestrating complex workflows and ensuring data consistency across different services.
- Seamless synchronization of data across microservices, improved reliability and scalability, and efficient handling of high-volume data changes.

3. Airbnb

Maintaining data consistency between primary databases and data warehouses for analytics.

- Airbnb uses Debezium, an open-source CDC tool, in combination with Apache Kafka to capture changes from their MySQL databases. These changes are then streamed to their data warehouse and analytical systems for real-time reporting and analysis.
- Real-time data availability for analytics, reduced latency in data processing, and enhanced decision-making capabilities based on up-to-date data.

Conclusion

Incorporating Change Data Capture (CDC) in system design ensures real-time data synchronization and supports event-driven architectures. CDC tracks changes in databases and promptly updates connected systems, maintaining data consistency and enabling responsive operations. It plays a crucial role in various applications, from real-time analytics to efficient data integration. By following best practices such as optimizing log-based tracking, managing schema changes, and ensuring fault tolerance, organizations can effectively handle large data volumes and maintain reliable, consistent data flows. Overall, CDC is essential for building dynamic, scalable, and resilient data systems.

let's unpack **data quality** and **data lineage**, and how tools like **Great Expectations** and **Monte Carlo** fit into the picture.

---

1. Data Quality: What & Why

Data quality means ensuring that your data is **accurate, complete, consistent, and reliable** for decision-making.

Key quality dimensions:

Dimension	**Meaning**	**Example issue**
Accuracy	Data matches reality	Price listed as 999 instead of 9.99
Completeness	No missing required values	Missing customer_id
Consistency	Data is uniform across systems	State field has "CA" and "California"
Timeliness	Data is up-to-date	Yesterday's sales report missing recent orders

Dimension	Meaning	Example issue
Validity	Conforms to expected format/rules	Date as 2025-13-40
Uniqueness	No duplicates unless expected	Duplicate invoices

---

2. Data Lineage: What & Why

Data lineage is the **map of data's journey** — how it moves and transforms from source to destination.

It answers:

- Where did this data come from? (source system)
- What transformations has it gone through? (ETL/ELT steps)
- Where is it used downstream? (dashboards, reports, ML models)

Benefits of lineage:

- Debugging issues (trace bad data back to source)
- Impact analysis before schema changes
- Compliance & auditing (GDPR, HIPAA)
- Trust building — users see where data came from

---

3. Great Expectations (Open Source)

Purpose: Data quality testing & documentation.

How it works:

1. **Expectations** — Define rules/tests for data ("expect column age to be between 0 and 120").
2. **Validation** — Run these expectations on datasets.
3. **Data Docs** — Generates human-readable validation reports.
4. **Checkpoints** — Bundles expectations and datasets into a scheduled validation run.

Example test:

```
import great_expectations as ge


df = ge.read_csv("orders.csv")

df.expect_column_values_to_be_between("order_amount", min_value=0)
```

df.expect_column_values_to_not_be_null("order_id")

df.expect_column_values_to_match_regex("email", r"[^@]+@[^@]+\.[^@]+")

Strengths:

- Open source, highly customizable.
- Integrates with batch jobs, Airflow, dbt.
- Good for **developer-defined** quality rules.

Limitations:

- Manual setup of expectations.
- Doesn't detect *unknown unknowns* — you must know what to check.

---

4. Monte Carlo (Commercial)

Purpose: Data observability platform — goes beyond testing to automatically monitor and alert.

How it works:

- **Monitors** data pipelines for anomalies in volume, freshness, and schema.
- **Lineage tracking** — maps dependencies between tables, jobs, and dashboards.
- **Incident detection & alerting** — notifies when data quality issues occur.
- **Integration** with warehouses (Snowflake, BigQuery), BI tools, and orchestration tools.

Example use case:

- Yesterday's orders table has 0 rows → Monte Carlo detects the volume drop → alerts Slack/Teams → shows lineage map so you see the upstream broken ingestion job.

Strengths:

- Auto-detection — no need to predefine every rule.
- Strong in **real-time anomaly detection**.
- Built-in lineage visualization.

Limitations:

- Paid SaaS product (per-table pricing).
- Black-box algorithms — less transparent than explicit expectations.

---

5. Great Expectations vs Monte Carlo — Quick Comparison

Feature	Great Expectations	Monte Carlo
Type	Data testing framework	Data observability platform
Rule definition	Manual, explicit	Automatic anomaly detection
Data lineage	No (needs integration)	Yes (built-in)
Alerting	Via integrations (Airflow, etc.)	Native real-time alerts
Cost	Open source (free)	Commercial (paid)
Ideal use	Known business/data rules	Catching unexpected issues

---

6. How They Work Together

A mature data stack often uses **both**:

- **Great Expectations** — to codify and enforce known quality rules (business logic).
- **Monte Carlo** — to catch unknown anomalies, monitor freshness, and visualize lineage.

Example flow:

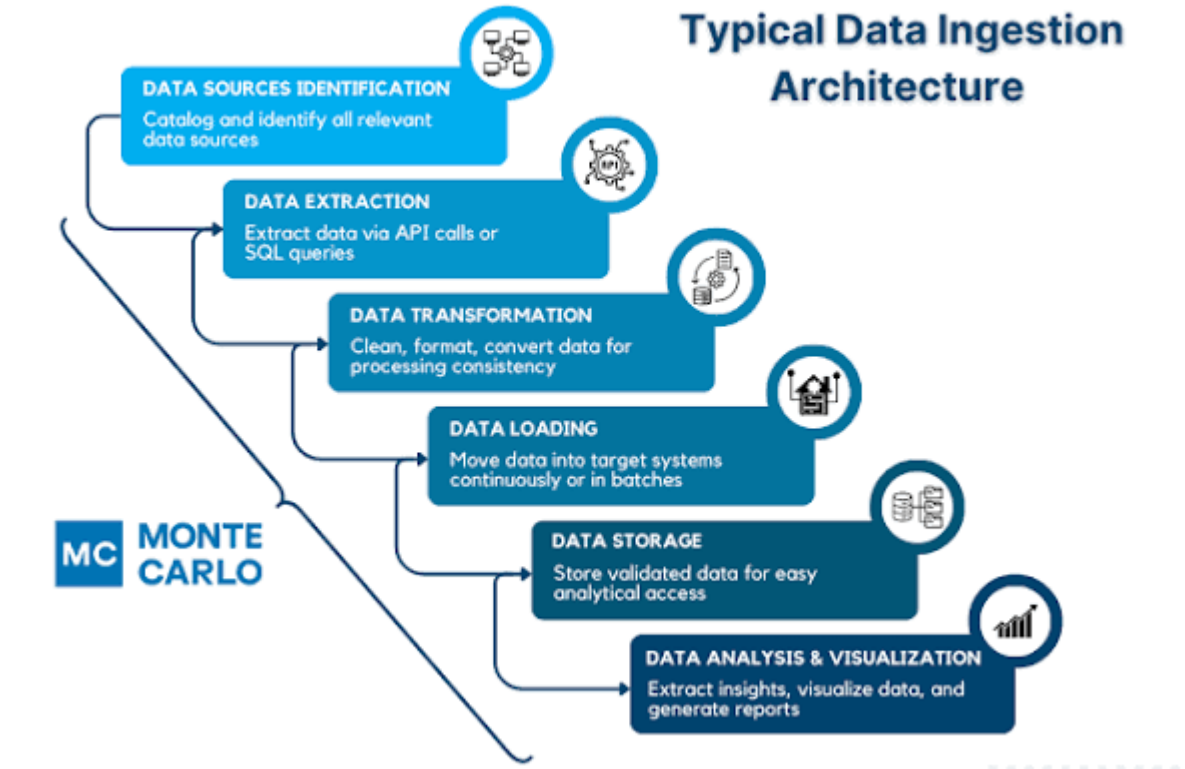
1. Raw data lands → **Monte Carlo** monitors volume/freshness anomalies.
2. Before loading into the warehouse → **Great Expectations** validates schema and value ranges.
3. Dashboards consuming this data → Monte Carlo lineage shows downstream impact if validation fails.

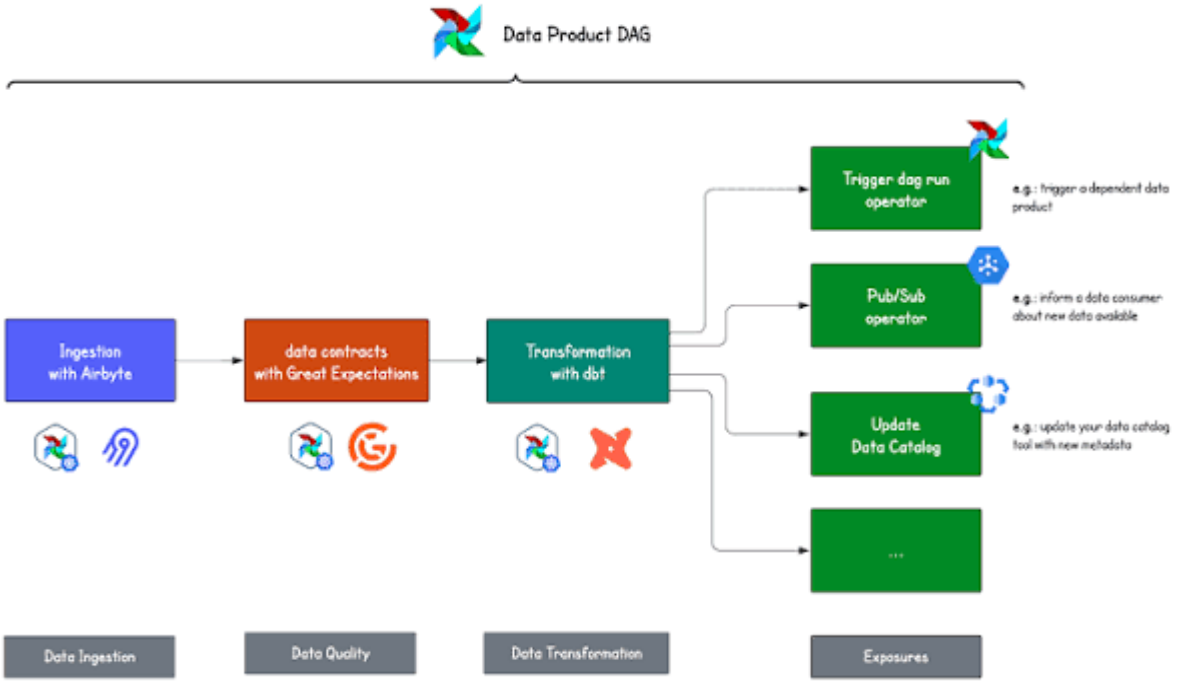
---

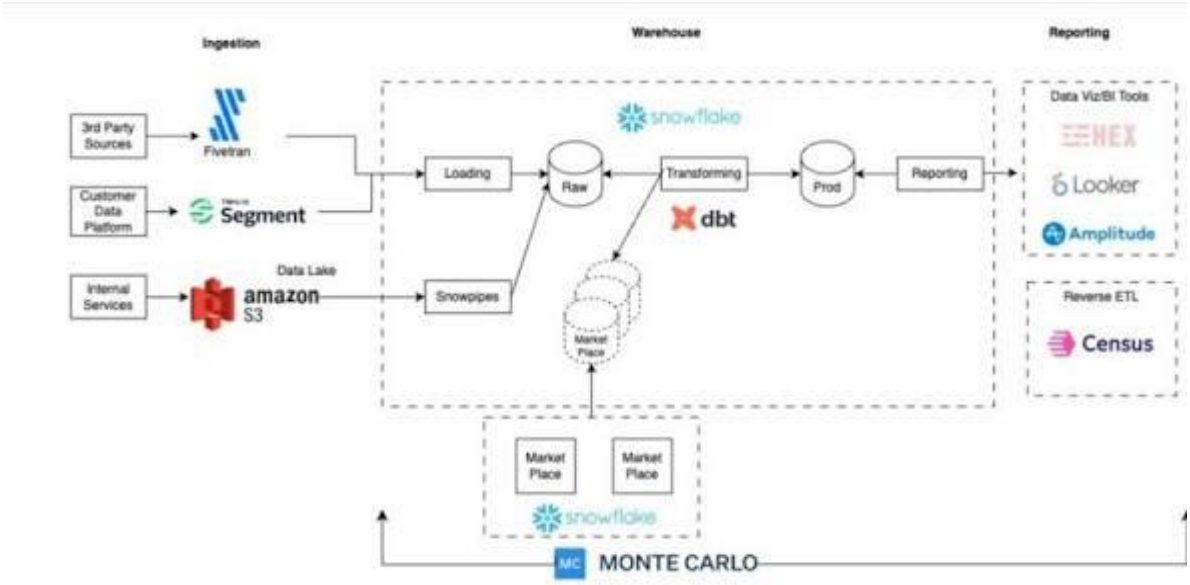
7. Best Practices for Data Quality & Lineage

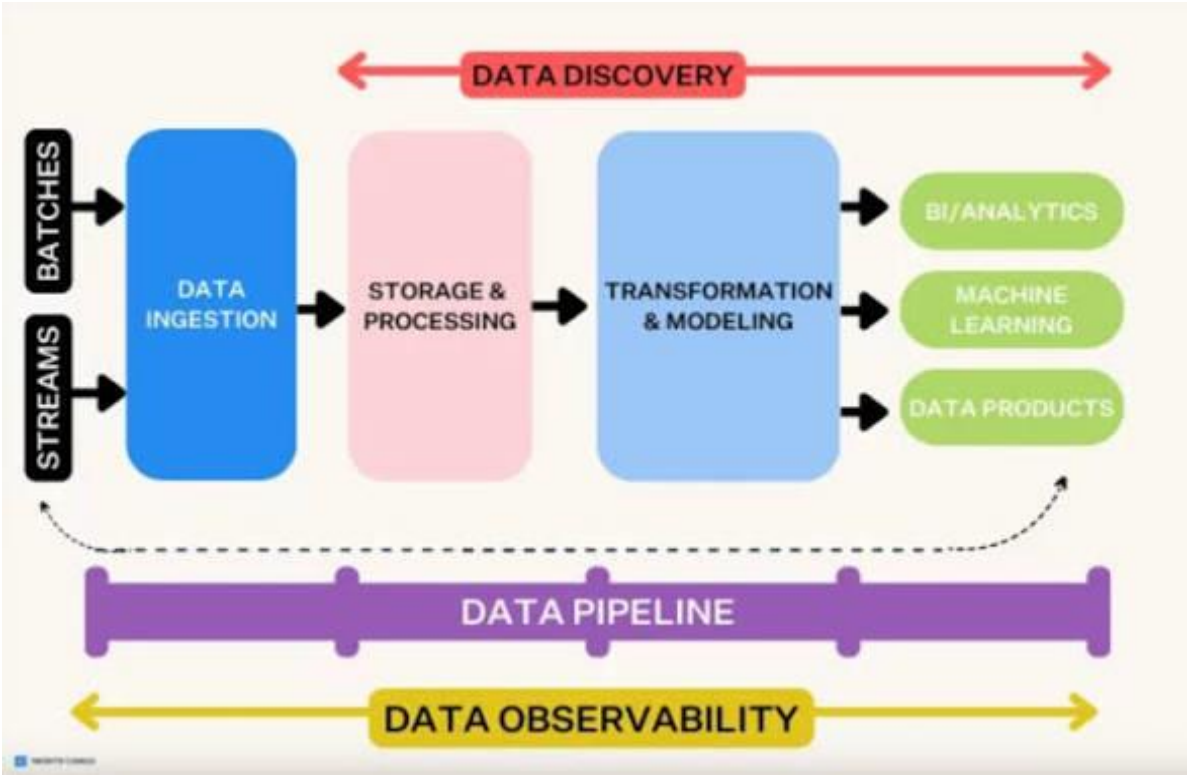
- **Automate checks** — run validations in pipelines (Airflow, dbt).
- **Monitor continuously** — detect drifts or pipeline delays.
- **Document lineage** — keep it up-to-date; automate when possible.
- **Fail fast** — stop bad data early before it spreads.
- **Communicate incidents** — alert data engineers & stakeholders quickly.
- **Track improvements** — log quality metrics over time to show progress.

Great Expectations and Monte Carlo integration with data pipeline tools

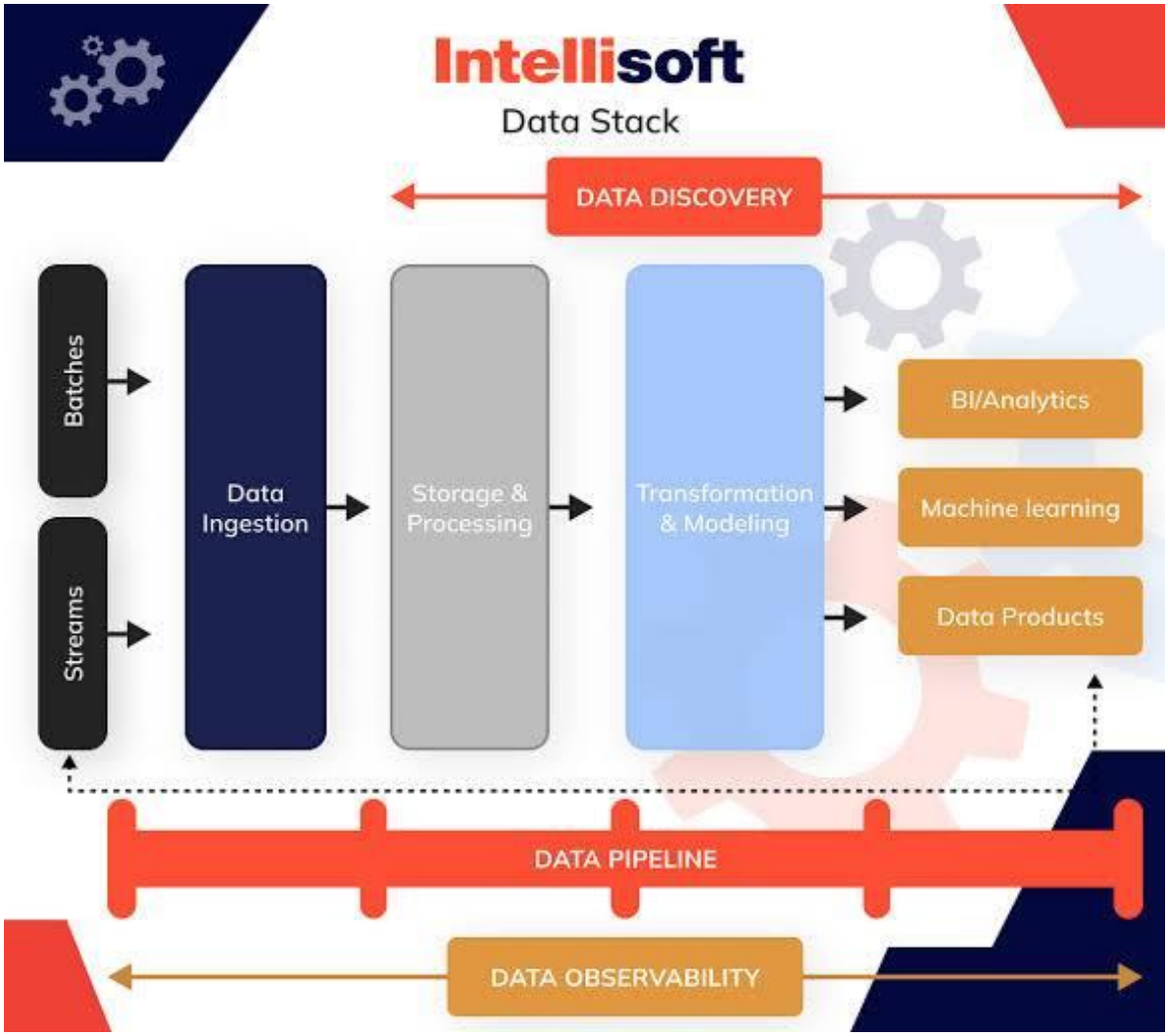








Ahmed Mohiuddin | LinkedIn



Modern data pipelines require robust mechanisms to ensure data quality and integrity at every stage. Great Expectations and Monte Carlo are two tools that can be effectively integrated with data ingestion, transformation, and orchestration layers to achieve this.

Architectural overview

Here's a conceptual overview of how Great Expectations and Monte Carlo integrate within a data pipeline architecture:

mermaid

graph TD

  subgraph Data Sources

    DS1(Database)

    DS2(API)

    DS3(Streaming Data)

  end

```
subgraph Data Ingestion
  DI(Ingestion Layer)
end

subgraph Data Transformation
  DT(Transformation Layer - ETL/ELT)
end

subgraph Data Storage
  DW(Data Warehouse/Lake)
end

subgraph Data Quality & Observability
  GE(Great Expectations)
  MC(Monte Carlo)
end

subgraph Data Orchestration
  DO(Orchestration Tool - e.g., Airflow, Prefect)
end

subgraph Monitoring & Alerting
  MA(Monitoring & Alerting)
end

subgraph Data Consumption
  BI(BI Tools/Analytics)
  ML(Machine Learning Models)
end

DS1 --> DI
```

DS2 --> DI

DS3 --> DI

DI --> DT

DT --> DW

DW --> GE

GE --> DO

DO --> GE

DW --> MC

MC --> MA

MC --> DO

DO --> DT

DT --> DW

DW --> BI

DW --> ML

GE --> MC

MC --> BI

MC --> ML

Use code with caution.

Explanation of components and integration

1. Data sources

- This represents various sources where data originates, such as relational databases (e.g., PostgreSQL, MySQL), APIs, and real-time streaming services (e.g., Apache Kafka, Amazon Kinesis.

2. Data ingestion

- The ingestion layer is responsible for collecting and importing data from diverse sources into the data pipeline.
- Tools like Apache NiFi, Apache Kafka, AWS Glue, and Hevo Data are commonly used for data ingestion.

3. Data transformation

- This layer involves processing and refining the ingested data to make it suitable for analysis and downstream consumption.
- ETL/ELT tools like dbt, Matillion, Talend, Informatica, and AWS Glue are used to perform transformations such as filtering, summarizing, and reformatting data.

4. Data storage

- Processed data is stored in data warehouses (e.g., Snowflake, BigQuery) or data lakes (e.g., AWS S3, Azure Data Lake).

5. Great expectations

- Great Expectations is an open-source Python library used for data validation, testing, and documentation.
- Integration with Data Transformation: Great Expectations can be integrated into the data transformation layer (e.g., within dbt models or Spark jobs) to validate data against predefined expectations before it's loaded into the data warehouse or lake.
- Creating Expectations: Data engineers define "expectations" – rules that specify what valid data should look like (e.g., column inventory_id should be non-null, part_num should be unique). These expectations are stored as JSON or YAML files.
- Data Validation: Great Expectations runs validation checks against the data, identifying deviations from expectations and generating detailed reports and data documentation.

6. Monte Carlo

- Monte Carlo is a data observability platform that provides end-to-end visibility into data pipelines.
- Integration with Data Storage: Monte Carlo connects to data warehouses and lakes to monitor metadata, detect anomalies, and track data lineage.
- Automated Monitoring and Alerting: Using machine learning, Monte Carlo automatically monitors data for freshness, volume, schema changes, and distribution anomalies, triggering alerts when issues arise.
- Data Lineage and Root Cause Analysis: Monte Carlo maps the flow of data from source to destination, helping identify the root cause of data quality issues when they occur.
- Integration with Great Expectations: Monte Carlo can leverage validation results from Great Expectations to enrich its data observability capabilities. It can use the insights from Great Expectations to refine its anomaly detection models and incident alerting.

7. Data orchestration

- Orchestration tools manage and automate the sequence of tasks in the data pipeline.
- Popular tools include Apache Airflow, dbt Cloud, Prefect, and Dagster.
- Integration with Great Expectations: Orchestration tools trigger the execution of Great Expectations checkpoints at various stages of the pipeline (e.g., after data ingestion, before data transformation, or before loading data into a specific table).
- Responding to Validation Results: Based on the validation results from Great Expectations, the orchestration tool can take actions such as halting the pipeline, notifying stakeholders, or triggering remedial processes.
- Integration with Monte Carlo: Monte Carlo can also integrate with orchestration tools to provide end-to-end lineage and enable incident resolution directly within the workflow.

8. Monitoring and alerting

- This layer provides dashboards and alerting mechanisms to visualize data quality metrics, track anomalies detected by Monte Carlo, and notify relevant teams or stakeholders about data issues.

9. Data consumption

- This layer represents the downstream systems that consume the data from the data warehouse or lake, such as Business Intelligence (BI) tools for reporting and analytics, and Machine Learning (ML) models.

Benefits of integration

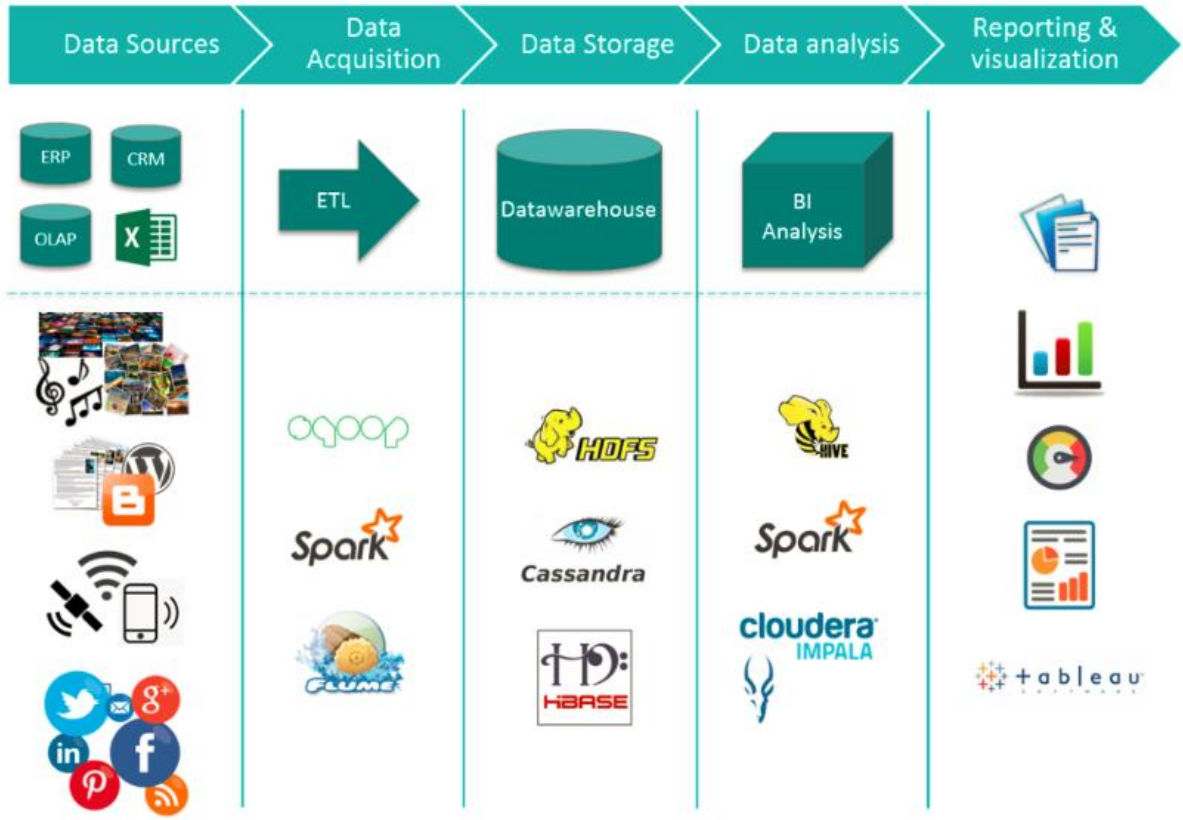
- Improved Data Quality: Great Expectations enforces data quality standards through validations, and Monte Carlo helps monitor data health and identify issues proactively.
- Early Detection of Issues: Integrating Great Expectations and Monte Carlo helps catch data quality problems early in the pipeline, preventing them from propagating downstream and affecting analytical insights or ML model performance.
- Increased Trust in Data: By providing automated validation, monitoring, and detailed reporting, the integrated solution builds trust in the data, empowering data-driven decision-making.
- Enhanced Data Observability: Monte Carlo provides end-to-end data observability, complementing Great Expectations' validation capabilities by offering a holistic view of data health and lineage across the entire pipeline.
- Streamlined Collaboration: By standardizing data validation and providing clear insights into data quality, the integration fosters better collaboration between data engineers, analysts, and business stakeholders.

By implementing such an architecture, organizations can achieve a more reliable, trustworthy, and efficient data pipeline, ensuring the quality of data that drives critical business insights and operations.

4. Big Data Ecosystem

Unit 5- Big Data Ecosystem - 06.05.18.

Unit 5- Big Data Ecosystem - 06.05.18.pdf



- **Hadoop, Hive, HDFS** – Fundamentals
- **Spark** (PySpark or Scala): RDD, DataFrame API, optimization, tuning
- **Kafka**: Streams, Connect, Kafka internals, offset management
- **Flink / Beam** (optional but good for solution architects)

Spark

Apache Spark is a powerful open-source, distributed processing system used for large-scale data processing and analytics. It offers flexible data processing capabilities through its APIs and allows for significant performance optimization through careful code writing and configuration tuning.

Here's a breakdown of the core concepts, APIs, and strategies for optimization and tuning in Spark:

1. RDD (Resilient Distributed Dataset)

- RDDs are Spark's fundamental data structure, representing an immutable, fault-tolerant collection of elements distributed across a cluster for parallel processing.
- They were the original Spark API and offer low-level control using functional programming.
- Benefits: Fine-grained control, fault-tolerance through lineage, immutability, and flexibility for unstructured/semi-structured data.
- Limitations: Lack of built-in optimizations, less developer-friendly, higher memory usage, and no automatic schema inference.
- Use Cases: Unstructured data, custom transformations, iterative algorithms, and graph processing.

2. DataFrame API

- DataFrames are a higher-level abstraction built on RDDs, similar to relational database tables with named columns and a defined schema.
- They improve developer experience and performance by leveraging Spark's internal optimizers.
- Benefits: High-level API with SQL-like capabilities, leverages Catalyst Optimizer, schema enforcement, faster and more memory-efficient than RDDs (due to optimizations like Tungsten), easier to use, and integrates with various data sources.
- Limitations: Lacks compile-time type safety, less flexible for low-level transformations than RDDs, and potential overhead for small-scale operations.
- Use Cases: Structured/semi-structured data processing, SQL-like queries, ETL, machine learning, and integration with BI/visualization tools.

3. Optimization and tuning strategies

Optimizing Spark applications is crucial for performance and efficiency. Key strategies include:

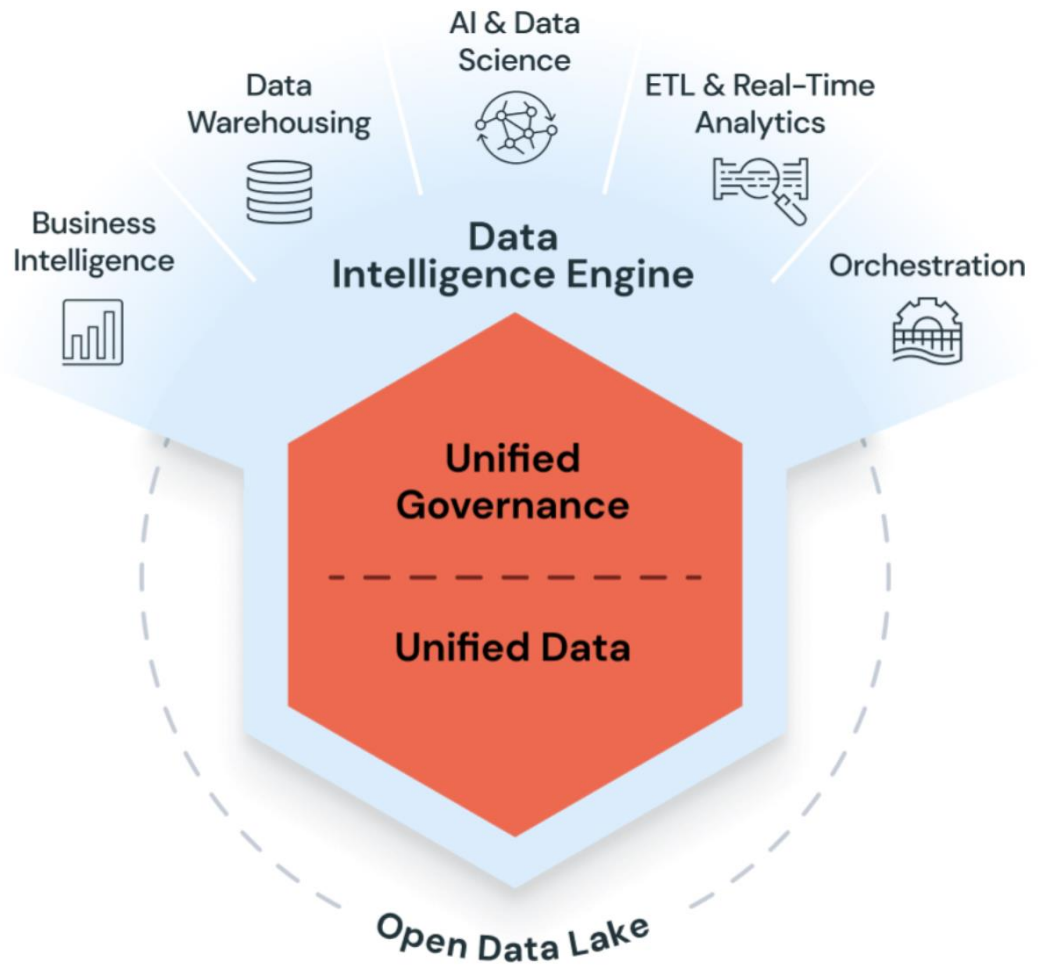
- Choose the Right API: Prioritize DataFrames/Datasets for built-in optimizations.
- Optimize Data Partitioning: Adjust partitions for even distribution and use formats like Parquet/ORC. Leverage predicate pushdown and partition pruning with formats like Parquet https://www.cloudthat.com/resources/blog/spark-dataframe-optimization-best-practices-and-techniques.
- Minimize Shuffle Operations: Shuffles are expensive. Filter data early, use reduceByKey() instead of groupByKey(), employ broadcast joins, and use map-side reductions.
- Utilize Built-in Functions Over UDFs: Spark's built-in functions are more performant. UDFs have overhead and aren't optimized by Catalyst. Consider Pandas UDFs in PySpark or Scala UDFs if necessary and minimize calls.
- Implement Effective Caching and Persistence: Cache intermediate data in memory or on disk using cache() or persist() to avoid recomputation.

- Leverage Adaptive Query Execution (AQE): AQE in Spark 3.0+ dynamically optimizes plans based on statistics. Enable it for dynamic join strategies, partition coalescing, and handling data skew https://www.chaosgenius.io/blog/spark-performance-tuning/.
- Tune Spark Configurations: Adjust memory and core settings, optimize garbage collection, and tune spark.sql.shuffle.partitions and spark.default.parallelism.
- Choose Efficient Data Formats: Use columnar formats like Parquet or ORC for better performance.
- Monitor and Analyze: Use the Spark UI and other tools to identify bottlenecks and iterate.

By understanding RDDs and DataFrames, using the right APIs, and applying these strategies, you can significantly enhance your Spark applications. Optimization is an ongoing process.

What is Azure Databricks?

Azure Databricks is a unified, open analytics platform for building, deploying, sharing, and maintaining enterprise-grade data, analytics, and AI solutions at scale. The Databricks Data Intelligence Platform integrates with cloud storage and security in your cloud account, and manages and deploys cloud infrastructure for you.



Azure Databricks uses generative AI with the data lakehouse to understand the unique semantics of your data. Then, it automatically optimizes performance and manages infrastructure to match your business needs.

Natural language processing learns your business's language, so you can search and discover data by asking a question in your own words. Natural language assistance helps you write code, troubleshoot errors, and find answers in documentation.

Managed open source integration

Databricks is committed to the open source community and manages updates of open source integrations with the Databricks Runtime releases. The following technologies are open source projects originally created by Databricks employees:

- Delta Lake and Delta Sharing
- MLflow
- Apache Spark and Structured Streaming
- Redash

- Unity Catalog

Common use cases

The following use cases highlight some of the ways customers use Azure Databricks to accomplish tasks essential to processing, storing, and analyzing the data that drives critical business functions and decisions.

Build an enterprise data lakehouse

The data lakehouse combines enterprise data warehouses and data lakes to accelerate, simplify, and unify enterprise data solutions. Data engineers, data scientists, analysts, and production systems can all use the data lakehouse as their single source of truth, providing access to consistent data and reducing the complexities of building, maintaining, and syncing many distributed data systems. See What is a data lakehouse?.

ETL and data engineering

Whether you're generating dashboards or powering artificial intelligence applications, data engineering provides the backbone for data-centric companies by making sure data is available, clean, and stored in data models for efficient discovery and use. Azure Databricks combines the power of Apache Spark with Delta and custom tools to provide an unrivaled ETL experience. Use SQL, Python, and Scala to compose ETL logic and orchestrate scheduled job deployment with a few clicks.

Lakeflow Declarative Pipelines further simplifies ETL by intelligently managing dependencies between datasets and automatically deploying and scaling production infrastructure to ensure timely and accurate data delivery to your specifications.

Azure Databricks provides tools for data ingestion, including Auto Loader, an efficient and scalable tool for incrementally and idempotently loading data from cloud object storage and data lakes into the data lakehouse.

Machine learning, AI, and data science

Azure Databricks machine learning expands the core functionality of the platform with a suite of tools tailored to the needs of data scientists and ML engineers, including MLflow and Databricks Runtime for Machine Learning.

Large language models and generative AI

Databricks Runtime for Machine Learning includes libraries like Hugging Face Transformers that allow you to integrate existing pre-trained models or other open source libraries into your workflow. The Databricks MLflow integration makes it easy to use the MLflow tracking service with transformer pipelines, models, and processing components. Integrate OpenAI models or solutions from partners like John Snow Labs in your Databricks workflows.

With Azure Databricks, customize a LLM on your data for your specific task. With the support of open source tooling, such as Hugging Face and DeepSpeed, you can efficiently take a foundation LLM and start training with your own data for more accuracy for your domain and workload.

In addition, Azure Databricks provides AI functions that SQL data analysts can use to access LLM models, including from OpenAI, directly within their data pipelines and workflows. See Apply AI on data using Azure Databricks AI Functions.

Data warehousing, analytics, and BI

Azure Databricks combines user-friendly UIs with cost-effective compute resources and infinitely scalable, affordable storage to provide a powerful platform for running analytic queries. Administrators configure scalable compute clusters as SQL warehouses, allowing end users to execute queries without worrying about any of the complexities of working in the cloud. SQL users can run queries against data in the lakehouse using the SQL query editor or in notebooks. Notebooks support Python, R, and Scala in addition to SQL, and allow users to embed the same visualizations available in legacy dashboards alongside links, images, and commentary written in markdown.

Data governance and secure data sharing

Unity Catalog provides a unified data governance model for the data lakehouse. Cloud administrators configure and integrate coarse access control permissions for Unity Catalog, and then Azure Databricks administrators can manage permissions for teams and individuals. Privileges are managed with access control lists (ACLs) through either user-friendly UIs or SQL syntax, making it easier for database administrators to secure access to data without needing to scale on cloud-native identity access management (IAM) and networking.

Unity Catalog makes running secure analytics in the cloud simple, and provides a division of responsibility that helps limit the reskilling or upskilling necessary for both administrators and end users of the platform. See What is Unity Catalog?.

The lakehouse makes data sharing within your organization as simple as granting query access to a table or view. For sharing outside of your secure environment, Unity Catalog features a managed version of Delta Sharing.

DevOps, CI/CD, and task orchestration

The development lifecycles for ETL pipelines, ML models, and analytics dashboards each present their own unique challenges. Azure Databricks allows all of your users to leverage a single data source, which reduces duplicate efforts and out-of-sync reporting. By additionally providing a suite of common tools for versioning, automating, scheduling, deploying code and production resources, you can simplify your overhead for monitoring, orchestration, and operations.

Jobs schedule Azure Databricks notebooks, SQL queries, and other arbitrary code. Databricks Asset Bundles allow you to define, deploy, and run Databricks resources such as jobs and pipelines programmatically. Git folders let you sync Azure Databricks projects with a number of popular git providers.

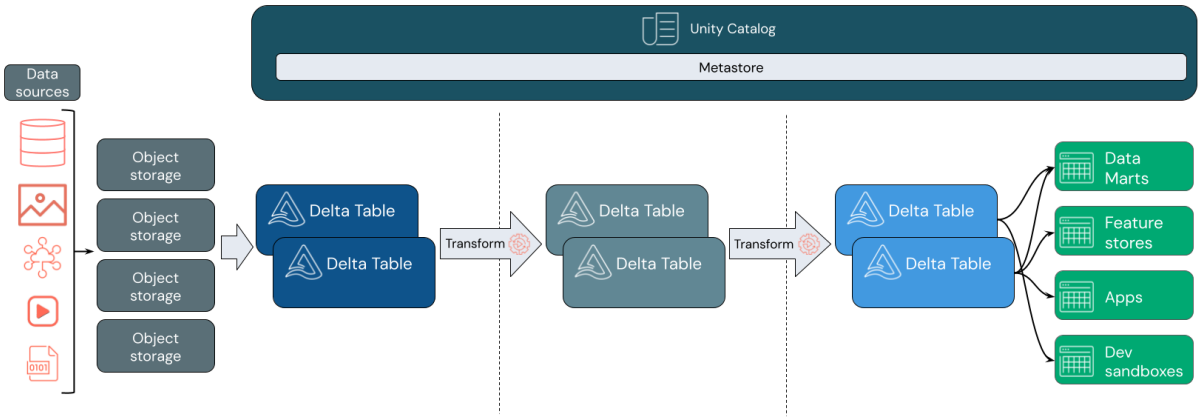
For CI/CD best practices and recommendations, see Best practices and recommended CI/CD workflows on Databricks. For a complete overview of tools for developers, see Develop on Databricks.

Real-time and streaming analytics

Azure Databricks leverages Apache Spark Structured Streaming to work with streaming data and incremental data changes. Structured Streaming integrates tightly with Delta Lake, and these technologies provide the foundations for both Lakeflow Declarative Pipelines and Auto Loader. See Structured Streaming concepts.

What is a data lakehouse?

A data lakehouse is a data management system that combines the benefits of data lakes and data warehouses. This article describes the lakehouse architectural pattern and what you can do with it on Azure Databricks.



What is a data lakehouse used for?

A data lakehouse provides scalable storage and processing capabilities for modern organizations that want to avoid isolated systems for processing different workloads, like machine learning (ML) and business intelligence (BI). A data lakehouse can help establish a single source of truth, eliminate redundant costs, and ensure data freshness.

Data lakehouses often use a data design pattern that incrementally improves, enriches, and refines data as it moves through layers of staging and transformation. Each layer of the lakehouse can include one or more layers. This pattern is frequently referred to as a medallion architecture. For more information, see What is the medallion lakehouse architecture?

How does the Databricks lakehouse work?

Databricks is built on Apache Spark. Apache Spark enables a massively scalable engine that runs on compute resources decoupled from storage. For more information, see Apache Spark on Azure Databricks

The Databricks lakehouse uses two additional key technologies:

- Delta Lake: an optimized storage layer that supports ACID transactions and schema enforcement.
- Unity Catalog: a unified, fine-grained governance solution for data and AI.

Data ingestion

At the ingestion layer, batch or streaming data arrives from a variety of sources and in a variety of formats. This first logical layer provides a place for that data to land in its raw format. As you convert those files to Delta tables, you can use the schema enforcement capabilities of Delta Lake to check for missing or unexpected data. You can use Unity Catalog to register tables according to your data governance model and required data isolation boundaries. Unity Catalog allows you to track the lineage of your data as it is transformed and refined, as well as apply a unified governance model to keep sensitive data private and secure.

Data processing, curation, and integration

Once verified, you can start curating and refining your data. Data scientists and machine learning practitioners frequently work with data at this stage to start combining or creating new features and complete data cleansing. Once your data has been thoroughly cleansed, it can be integrated and reorganized into tables designed to meet your particular business needs.

A schema-on-write approach, combined with Delta schema evolution capabilities, means that you can make changes to this layer without necessarily having to rewrite the downstream logic that serves data to your end users.

Data serving

The final layer serves clean, enriched data to end users. The final tables should be designed to serve data for all your use cases. A unified governance model means you can track data lineage back to your single source of truth. Data layouts, optimized for different tasks, allow end users to access data for machine learning applications, data engineering, and business intelligence and reporting.

To learn more about Delta Lake, see What is Delta Lake in Azure Databricks? To learn more about Unity Catalog, see What is Unity Catalog?

Capabilities of a Databricks lakehouse

A lakehouse built on Databricks replaces the current dependency on data lakes and data warehouses for modern data companies. Some key tasks you can perform include:

- **Real-time data processing:** Process streaming data in real-time for immediate analysis and action.
- **Data integration:** Unify your data in a single system to enable collaboration and establish a single source of truth for your organization.
- **Schema evolution:** Modify data schema over time to adapt to changing business needs without disrupting existing data pipelines.
- **Data transformations:** Using Apache Spark and Delta Lake brings speed, scalability, and reliability to your data.
- **Data analysis and reporting:** Run complex analytical queries with an engine optimized for data warehousing workloads.
- **Machine learning and AI:** Apply advanced analytics techniques to all of your data. Use ML to enrich your data and support other workloads.

- **Data versioning and lineage:** Maintain version history for datasets and track lineage to ensure data provenance and traceability.
- **Data governance:** Use a single, unified system to control access to your data and perform audits.
- **Data sharing:** Facilitate collaboration by allowing the sharing of curated data sets, reports, and insights across teams.
- **Operational analytics:** Monitor data quality metrics, model quality metrics, and drift by applying machine learning to lakehouse monitoring data.

Lakehouse vs Data Lake vs Data Warehouse

Data warehouses have powered business intelligence (BI) decisions for about 30 years, having evolved as a set of design guidelines for systems controlling the flow of data. Enterprise data warehouses optimize queries for BI reports, but can take minutes or even hours to generate results. Designed for data that is unlikely to change with high frequency, data warehouses seek to prevent conflicts between concurrently running queries. Many data warehouses rely on proprietary formats, which often limit support for machine learning. Data warehousing on Azure Databricks leverages the capabilities of a Databricks lakehouse and Databricks SQL. For more information, see Data warehousing on Azure Databricks.

Powered by technological advances in data storage and driven by exponential increases in the types and volume of data, data lakes have come into widespread use over the last decade. Data lakes store and process data cheaply and efficiently. Data lakes are often defined in opposition to data warehouses: A data warehouse delivers clean, structured data for BI analytics, while a data lake permanently and cheaply stores data of any nature in any format. Many organizations use data lakes for data science and machine learning, but not for BI reporting due to its unvalidated nature.

The data lakehouse combines the benefits of data lakes and data warehouses and provides:

- Open, direct access to data stored in standard data formats.
- Indexing protocols optimized for machine learning and data science.
- Low query latency and high reliability for BI and advanced analytics.

By combining an optimized metadata layer with validated data stored in standard formats in cloud object storage, the data lakehouse allows data scientists and ML engineers to build models from the same data-driven BI reports.

What are the Delta things used for?

Delta is a term introduced with Delta Lake, the foundation for storing data and tables in the Databricks lakehouse. Delta Lake was conceived of as a unified data management system for handling transactional real-time and batch big data, by extending Parquet data files with a file-based transaction log for ACID transactions and scalable metadata handling.

Delta Lake: OS data management for the lakehouse

Delta Lake is an open-source storage layer that brings reliability to data lakes by adding a transactional storage layer on top of data stored in cloud storage (on AWS S3, Azure Storage, and GCS). It allows for ACID transactions, data versioning, and rollback capabilities. It allows you to handle both batch and streaming data in a unified way.

Delta tables are built on top of this storage layer and provide a table abstraction, making it easy to work with large-scale structured data using SQL and the DataFrame API.

Delta tables: Default data table architecture

Delta table is the default data table format in Azure Databricks and is a feature of the Delta Lake open source data framework. Delta tables are typically used for data lakes, where data is ingested via streaming or in large batches.

Lakeflow Declarative Pipelines: Data pipelines

Lakeflow Declarative Pipelines manage the flow of data between many Delta tables, thus simplifying the work of data engineers on ETL development and management. The pipeline is the main unit of execution for Lakeflow Declarative Pipelines. Lakeflow Declarative Pipelines offers declarative pipeline development, improved data reliability, and cloud-scale production operations. Users can perform both batch and streaming operations on the same table and the data is immediately available for querying. You define the transformations to perform on your data, and Lakeflow Declarative Pipelines manages task orchestration, cluster management, monitoring, data quality, and error handling. Lakeflow Declarative Pipelines enhanced autoscaling can handle streaming workloads which are spiky and unpredictable.

See the Lakeflow Declarative Pipelines tutorial.

Delta tables vs. Lakeflow Declarative Pipelines

Delta table is a way to store data in tables, whereas Lakeflow Declarative Pipelines allows you to describe how data flows between these tables declaratively. Lakeflow Declarative Pipelines is a declarative framework that manages many delta tables, by creating them and keeping them up to date. In short, Delta tables is a data table architecture while Lakeflow Declarative Pipelines is a data pipeline framework.

Delta: Open source or proprietary?

A strength of the Azure Databricks platform is that it doesn't lock customers into proprietary tools: Much of the technology is powered by open source projects, which Azure Databricks contributes to.

The Delta OSS projects are examples:

- Delta Lake project: Open source storage for a lakehouse.
- Delta Sharing protocol: Open protocol for secure data sharing.

Lakeflow Declarative Pipelines is a proprietary framework in Azure Databricks.

What are the other *Delta* things on Azure Databricks?

Below are descriptions of other features that include *Delta* in their name.

Delta Sharing

An open standard for secure data sharing, Delta Sharing enables data sharing between organizations regardless of their compute platform.

Delta engine

A query optimizer for big data that uses Delta Lake open source technology included in Databricks. Delta engine optimizes the performance of Spark SQL, Databricks SQL, and DataFrame operations by pushing computation to the data.

Delta Lake transaction log (AKA DeltaLogs)

A single source of truth tracking all changes that users make to the table and the mechanism through which Delta Lake guarantees atomicity. See the Delta transaction log protocol on GitHub.

The transaction log is key to understanding Delta Lake, because it is the common thread that runs through many of its most important features:

- ACID transactions
- Scalable metadata handling
- Time travel
- And more.

Azure Databricks components

Accounts and workspaces

In Azure Databricks, a *workspace* is an Azure Databricks deployment in the cloud that functions as an environment for your team to access Databricks assets. Your organization can choose to have either multiple workspaces or just one, depending on its needs.

An Azure Databricks *account* represents a single entity that can include multiple workspaces. Accounts enabled for Unity Catalog can be used to manage users and their access to data centrally across all of the workspaces in the account.

Billing: Databricks units (DBUs)

Azure Databricks bills based on Databricks units (DBUs), which are units of processing capability per hour based on VM instance type.

Authentication and authorization

This section describes concepts that you need to know when you manage Azure Databricks identities and their access to Azure Databricks assets.

User

A unique individual who has access to the system. User identities are represented by email addresses. See Manage users.

Service principal

A service identity for use with jobs, automated tools, and systems such as scripts, apps, and CI/CD platforms. Service principals are represented by an application ID. See Service principals.

Group

A collection of identities. Groups simplify identity management, making it easier to assign access to workspaces, data, and other securable objects. All Databricks identities can be assigned as members of groups. See Groups.

Access control list (ACL)

A list of permissions attached to the workspace, cluster, job, table, or experiment. An ACL specifies which users or system processes are granted access to the objects, as well as what operations are allowed on the assets. Each entry in a typical ACL specifies a subject and an operation. See Access control lists.

Personal access token (PAT)

A personal access token is a string used to authenticate REST API calls, Technology partners connections, and other tools. See Azure Databricks personal access token authentication.

Microsoft Entra ID tokens can also be used to authenticate to the REST API.

Azure Databricks interfaces

This section describes the interfaces for accessing your assets in Azure Databricks.

UI

The Azure Databricks UI is a graphical interface for interacting with features, such as workspace folders and their contained objects, data objects, and computational resources.

REST API

The Databricks REST API provides endpoints for modifying or requesting information about Azure Databricks account and workspace objects. See account reference and workspace reference.

SQL REST API

The SQL REST API allows you to automate tasks on SQL objects. See SQL API.

CLI

The Databricks CLI is hosted on GitHub. The CLI is built on top of the Databricks REST API.

Data management

This section describes the tools and logical objects used to organize and govern data on Azure Databricks. See Database objects in Azure Databricks.

Unity Catalog

Unity Catalog is a unified governance solution for data and AI assets on Azure Databricks that provides centralized access control, auditing, lineage, and data discovery capabilities across Databricks workspaces. See What is Unity Catalog?.

Catalog

Catalogs are the highest level container for organizing and isolating data on Azure Databricks. You can share catalogs across workspaces within the same region and account. See What are catalogs in Azure Databricks?.

Schema

Schemas, also known as databases, are contained within catalogs and provide a more granular level of organization. They contain database objects and AI assets, such as volumes, tables, functions, and models. See What are schemas in Azure Databricks?.

Table

Tables organize and govern access to structured data. You query tables with Apache Spark SQL and Apache Spark APIs. See Introduction to Azure Databricks tables.

View

A view is a read-only object derived from one or more tables and views. Views save queries that are defined against tables. See What is a view?.

Volume

Volumes represent a logical volume of storage in a cloud object storage location and organize and govern access to non-tabular data. Databricks recommends using volumes for managing all access to non-tabular data on cloud object storage. See What are Unity Catalog volumes?.

Delta table

By default, all tables created in Azure Databricks are Delta tables. Delta tables are based on the Delta Lake open source project, a framework for high-performance ACID table storage over cloud object stores. A Delta table stores data as a directory of files on cloud object storage and registers table metadata to the metastore within a catalog and schema.

Find out more about technologies branded as Delta.

Metastore

Unity Catalog provides an account-level metastore that registers metadata about data, AI, and permissions about catalogs, schemas, and tables. See Metastore.

Azure Databricks provides a legacy Hive metastore for customers that have not adopted Unity Catalog. See Hive metastore table access control (legacy).

Catalog Explorer

Catalog Explorer allows you to explore and manage data and AI assets, including schemas (databases), tables, models, volumes (non-tabular data), functions, and registered ML models. You can use it to find data objects and owners, understand data relationships across tables, and manage permissions and sharing. See What is Catalog Explorer?.

Computation management

This section describes concepts that you need to know to run computations in Azure Databricks.

Cluster

A set of computation resources and configurations on which you run notebooks and jobs. There are two types of clusters: all-purpose and job. See Compute.

- You create an *all-purpose cluster* using the UI, CLI, or REST API. You can manually terminate and restart an all-purpose cluster. Multiple users can share such clusters to do collaborative interactive analysis.
- The Azure Databricks job scheduler creates *a job cluster* when you run a job on a *new job cluster* and terminates the cluster when the job is complete. You *cannot* restart an job cluster.

Pool

A set of idle, ready-to-use instances that reduce cluster start and auto-scaling times. When attached to a pool, a cluster allocates its driver and worker nodes from the pool. See Pool configuration reference.

If the pool does not have sufficient idle resources to accommodate the cluster's request, the pool expands by allocating new instances from the instance provider. When an attached cluster is terminated, the instances it used are returned to the pool and can be reused by a different cluster.

Databricks runtime

The set of core components that run on the clusters managed by Azure Databricks.
See Compute. Azure Databricks has the following runtimes:

- Databricks Runtime includes Apache Spark but also adds a number of components and updates that substantially improve the usability, performance, and security of big data analytics.
- Databricks Runtime for Machine Learning is built on Databricks Runtime and provides prebuilt machine learning infrastructure that is integrated with all of the capabilities of the Azure Databricks workspace. It contains multiple popular libraries, including TensorFlow, Keras, PyTorch, and XGBoost.

Jobs & Pipelines UI

The **Jobs & Pipelines** workspace UI provides entry to the Jobs, Lakeflow Declarative Pipelines, and Lakeflow Connect UIs, which are tools that allow you to orchestrate and schedule workflows.

Jobs

A non-interactive mechanism for orchestrating and scheduling notebooks, libraries, and other tasks. See Lakeflow Jobs

Pipelines

Lakeflow Declarative Pipelines provide a declarative framework for building reliable, maintainable, and testable data processing pipelines. See Lakeflow Declarative Pipelines.

Workload

Workload is the amount of processing capability needed to perform a task or group of tasks. Azure Databricks identifies two types of workloads: data engineering (job) and data analytics (all-purpose).

- **Data engineering** An (automated) workload runs on *a job cluster* which the Azure Databricks job scheduler creates for each workload.
- **Data analytics** An (interactive) workload runs on an *all-purpose cluster*. Interactive workloads typically run commands within an Azure Databricks notebook. However, running a *job* on an *existing all-purpose* cluster is also treated as an interactive workload.

Execution context

The state for a read–eval–print loop (REPL) environment for each supported programming language. The languages supported are Python, R, Scala, and SQL.

Data engineering

Data engineering tools aid collaboration among data scientists, data engineers, data analysts, and machine learning engineers.

Workspace

A workspace is an environment for accessing all of your Azure Databricks assets. A workspace organizes objects (notebooks, libraries, dashboards, and experiments) into folders and provides access to data objects and computational resources.

Notebook

A web-based interface for creating data science and machine learning workflows that can contain runnable commands, visualizations, and narrative text. See Databricks notebooks.

Library

A package of code available to the notebook or job running on your cluster. Databricks runtimes include many libraries, and you can also upload your own. See Install libraries.

Git folder (formerly Repos)

A folder whose contents are co-versioned together by syncing them to a remote Git repository. Databricks Git folders integrate with Git to provide source and version control for your projects.

AI and machine learning

Databricks provides an integrated end-to-end environment with managed services for developing and deploying AI and machine learning applications.

Mosaic AI

The brand name for products and services from Databricks Mosaic AI Research, a team of researchers and engineers responsible for Databricks biggest breakthroughs in generative AI. Mosaic AI products include the ML and AI features in Databricks. See Mosaic Research.

Machine learning runtime

To help you develop ML and AI models, Databricks provides a Databricks Runtime for Machine Learning, which automates compute creation with pre-built machine learning and deep learning infrastructure including the most common ML and DL libraries. It also has built-in, pre-configured GPU support including drivers and supporting libraries. Browse to information about the latest runtime releases from Databricks Runtime release notes versions and compatibility.

Experiment

A collection of MLflow runs for training a machine learning model. See Organize training runs with MLflow experiments.

Features

Features are an important component of ML models. A feature store enables feature sharing and discovery across your organization and also ensures that the same feature computation code is used for model training and inference. See Feature management.

Generative AI models

Databricks supports the exploration, development, and deployment of generative AI models, including:

- AI playground, a chat-like environment in the workspace where you can test, prompt, and compare LLMs. See Chat with LLMs and prototype generative AI apps using AI Playground.
- A built-in set of pre-configured foundation models that you can query:
    - See Pay-per-token Foundation Model APIs.
    - See [Recommended] Deploy foundation models from Unity Catalog for foundation models you can serve with a single click.
- Third-party hosted LLMs, called external models. These models are meant to be used as-is.
- Capabilities to customize a foundation model to optimize its performance for your specific application (often called fine-tuning). See Foundation Model Fine-tuning.

Model registry

Databricks provides a hosted version of MLflow Model Registry in Unity Catalog. Models registered in Unity Catalog inherit centralized access control, lineage, and cross-workspace discovery and access. See Manage model lifecycle in Unity Catalog.

Model serving

Mosaic AI Model Serving provides a unified interface to deploy, govern, and query AI models. Each model you serve is available as a REST API that you can integrate into your web or client application. With Mosaic AI Model Serving, you can deploy your own models, foundation

models, or third-party models hosted outside of Databricks. See Deploy models using Mosaic AI Model Serving.

Data warehousing

Data warehousing refers to collecting and storing data from multiple sources so it can be quickly accessed for business insights and reporting. Databricks SQL is the collection of services that bring data warehousing capabilities and performance to your existing data lakes. See Data warehousing architecture.

Query

A query is a valid SQL statement that allows you to interact with your data. You can author queries using the in-platform SQL editor, or connect using a SQL connector, driver, or API. See Access and manage saved queries to learn more about how to work with queries.

SQL warehouse

A computation resource on which you run SQL queries. There are three types of SQL warehouses: Classic, Pro, and Serverless. Azure Databricks recommends using serverless warehouses where available. See SQL warehouse types to compare available features for each warehouse type.

Query history

A list of executed queries and their performance characteristics. Query history allows you to monitor query performance, helping you identify bottlenecks and optimize query runtimes. See Query history.

Visualization

A graphical presentation of the result of running a query. See Visualizations in Databricks notebooks and SQL editor.

Dashboard

A presentation of data visualizations and commentary. You can use dashboards to automatically send reports to anyone in your Azure Databricks account. Use the Databricks Assistant to help you build visualizations based on natural language prompts. See Dashboards. You can also create a dashboard from a notebook. See Dashboards in notebooks.

Azure Databricks integrations overview

The articles listed here provide information about how to connect to the large assortment of data sources, BI tools, and developer tools that you can use with Azure Databricks. Many of these are available through our system of partners and our Partner Connect hub.

Partner Connect

Partner Connect is a user interface that allows validated solutions to integrate more quickly and easily with your Databricks clusters and SQL warehouses.

For more information, see What is Databricks Partner Connect?.

Data sources

Databricks can read data from and write data to a variety of data formats such as CSV, Delta Lake, JSON, Parquet, XML, and other formats, as well as data storage providers such as Amazon S3, Google BigQuery and Cloud Storage, Snowflake, and other providers.

See Data ingestion, Connect to data sources and external services, and Data format options.

BI tools

Databricks has validated integrations with your favorite BI tools, including Power BI, Tableau, and others, allowing you to work with data through Databricks clusters and SQL warehouses, in many cases with low-code and no-code experiences.

For a comprehensive list, with connection instructions, see BI and visualization.

Other ETL tools

In addition to access to all kinds of data sources, Databricks provides integrations with ETL/ELT tools like dbt, Prophecy, and Azure Data Factory, as well as data pipeline orchestration tools like Airflow and SQL database tools like DataGrip, DBeaver, and SQL Workbench/J.

For connection instructions, see:

- **ETL tools**: Data preparation and transformation
- **Airflow**: Orchestrate Lakeflow Jobs with Apache Airflow
- **SQL database tools**: SQL connectors, libraries, drivers, APIs, and tools.

IDEs and other developer tools

Databricks supports developer tools such as DataGrip, IntelliJ, PyCharm, Visual Studio Code, and others, that allow you to programmatically access Azure Databricks compute, including SQL warehouses.

For a comprehensive list of tools that support developers, see Develop on Databricks.

Git

Databricks Git folders provide repository-level integration with your favorite Git providers, so you can develop code in a Databricks notebook and sync it with a remote Git repository. See What is Databricks Git folders.

Azure Databricks architecture overview

This article provides a high-level overview of Azure Databricks architecture, including its enterprise architecture, in combination with Azure.

High-level architecture

Azure Databricks operates out of a **control plane** and a **compute plane**.

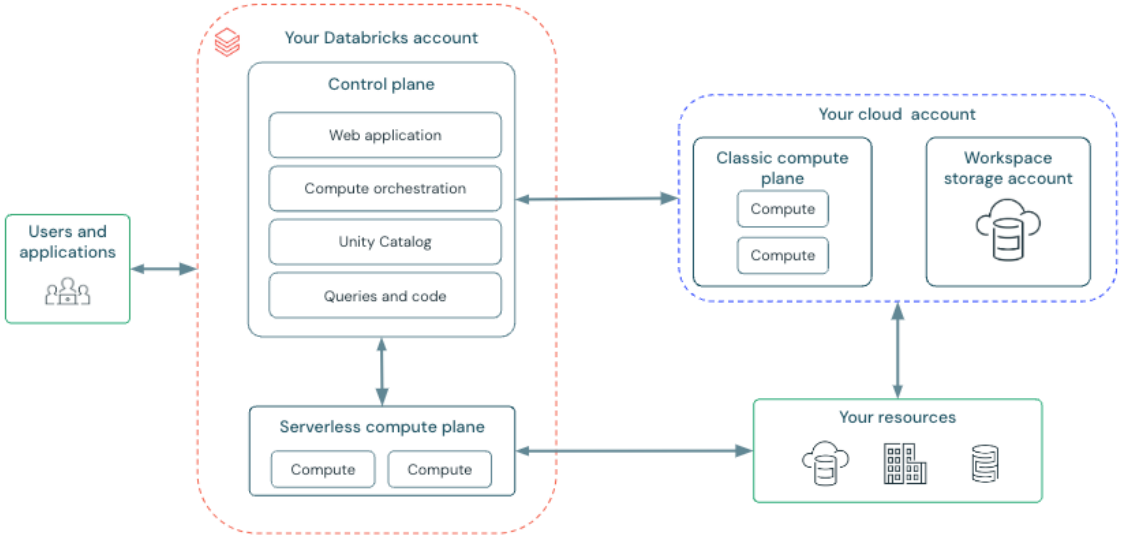
- The **control plane** includes the backend services that Azure Databricks manages in your Azure Databricks account. The web application is in the control plane.
- The **compute plane** is where your data is processed. There are two types of compute planes depending on the compute that you are using.

- For serverless compute, the serverless compute resources run in a *serverless compute plane* in your Azure Databricks account.
- For classic Azure Databricks compute, the compute resources are in your Azure subscription in what is called the *classic compute plane*. This refers to the network in your Azure subscription and its resources.

To learn more about classic compute and serverless compute, see Types of compute.

Each Azure Databricks workspace has an associated storage account known as the **workspace storage account**. The workspace storage account is in your Azure subscription.

The following diagram describes the overall Azure Databricks architecture.



Serverless compute plane

In the serverless compute plane, Azure Databricks compute resources run in a compute layer within your Azure Databricks account. Azure Databricks creates a serverless compute plane in the same Azure region as your workspace's classic compute plane. You select this region when creating a workspace.

To protect customer data within the serverless compute plane, serverless compute runs within a network boundary for the workspace, with various layers of security to isolate different Azure Databricks customer workspaces and additional network controls between clusters of the same customer.

To learn more about networking in the serverless compute plane, Serverless compute plane networking.

Classic compute plane

In the classic compute plane, Azure Databricks compute resources run in your Azure subscription. New compute resources are created within each workspace's virtual network in the customer's Azure subscription.

A classic compute plane has natural isolation because it runs in each customer's own Azure subscription. To learn more about networking in the classic compute plane, see Classic compute plane networking.

For regional support, see Azure Databricks regions.

Workspace storage account

When you create a workspace, Azure Databricks creates a account in your Azure subscription to use as the workspace storage account.

The workspace storage account contains:

- **Workspace system data:** Workspace system data is generated as you use various Azure Databricks features such as creating notebooks. This bucket includes notebook revisions, job run details, command results, and Spark logs
- **DBFS:** DBFS (Databricks File System) is a distributed file system in Azure Databricks environments accessible under the dbfs:/ namespace. DBFS root and DBFS mounts are both in the dbfs:/ namespace. Storing and accessing data using DBFS root or DBFS mounts is a deprecated pattern and not recommended by Databricks. For more information, see What is DBFS?.
- **Unity Catalog workspace catalog:** If your workspace was enabled for Unity Catalog automatically, the workspace storage account contains the default workspace catalog. All users in your workspace can create assets in the default schema in this catalog. See Get started with Unity Catalog.

What are ACID guarantees on Azure Databricks?

Azure Databricks uses Delta Lake by default for all reads and writes and builds upon the ACID guarantees provided by the open source Delta Lake protocol. ACID stands for atomicity, consistency, isolation, and durability.

- Atomicity means that all transactions either succeed or fail completely.
- Consistency guarantees relate to how a given state of the data is observed by simultaneous operations.
- Isolation refers to how simultaneous operations potentially conflict with one another.
- Durability means that committed changes are permanent.

While many data processing and warehousing technologies describe having ACID transactions, specific guarantees vary by system, and transactions on Azure Databricks might differ from other systems you've worked with.

How are transactions scoped on Azure Databricks?

Azure Databricks manages transactions at the table level. Transactions always apply to one table at a time. For managing concurrent transactions, Azure Databricks uses optimistic concurrency control. This means that there are no locks on reading or writing against a table, and deadlock is not a possibility.

By default, Azure Databricks provides snapshot isolation on reads and write-serializable isolation on writes. Write-serializable isolation provides stronger guarantees than snapshot isolation, but it applies that stronger isolation only for writes.

Read operations referencing multiple tables return the current version of each table at the time of access, but do not interrupt concurrent transactions that might modify referenced tables.

Azure Databricks does not have BEGIN/END constructs that allow multiple operations to be grouped together as a single transaction. Applications that modify multiple tables commit transactions to each table in a serial fashion. You can combine inserts, updates, and deletes against a table into a single write transaction using MERGE INTO.

How does Azure Databricks implement atomicity?

The transaction log controls commit atomicity. During a transaction, data files are written to the file directory backing the table. When the transaction completes, a new entry is committed to the transaction log that includes the paths to all files written during the transaction. Each commit increments the table version and makes new data files visible to read operations. The current state of the table comprises all data files marked valid in the transaction logs.

Data files are not tracked unless the transaction log records a new version. If a transaction fails after writing data files to a table, these data files will not corrupt the table state, but the files will not become part of the table. The VACUUM operation deletes all untracked data files in a table directory, including remaining uncommitted files from failed transactions.

How does Azure Databricks implement durability?

Azure Databricks uses cloud object storage to store all data files and transaction logs. Cloud object storage has high availability and durability. Because transactions either succeed or fail completely and the transaction log lives alongside data files in cloud object storage, tables on Azure Databricks inherit the durability guarantees of the cloud object storage on which they're stored.

How does Azure Databricks implement consistency?

Delta Lake uses optimistic concurrency control to provide transactional guarantees between writes. Under this mechanism, writes operate in three stages:

1. **Read**: Reads (if needed) the latest available version of the table to identify which files need to be modified (that is, rewritten).
    - Writes that are append-only do not read the current table state before writing. Schema validation leverages metadata from the transaction log.
2. **Write**: Writes data files to the directory used to define the table.
3. **Validate and commit**:
    - Checks whether the proposed changes conflict with any other changes that may have been concurrently committed since the snapshot that was read.
    - If there are no conflicts, all the staged changes are committed as a new versioned snapshot, and the write operation succeeds.

- If there are conflicts, the write operation fails with a concurrent modification exception. This failure prevents corruption of data.

Optimistic concurrency assumes that most concurrent transactions on your data could not conflict with one another, but conflicts can occur. See Isolation levels and write conflicts on Azure Databricks.

How does Azure Databricks implement isolation?

Azure Databricks uses write serializable isolation by default for all table writes and updates. Snapshot isolation is used for all table reads.

Write serializability and optimistic concurrency control work together to provide high throughput for writes. The current valid state of a table is always available, and a write can be started against a table at any time. Concurrent reads are only limited by throughput of the metastore and cloud resources.

Does Delta Lake support multi-table transactions?

Delta Lake does not support multi-table transactions. Delta Lake supports transactions at the *table* level.

Primary key and foreign key relationships on Azure Databricks are informational and not enforced. See Declare primary key and foreign key relationships.

What does it mean that Delta Lake supports multi-cluster writes?

Delta Lake prevents data corruption when multiple clusters write to the same table concurrently. Some write operations can conflict during simultaneous execution, but don't corrupt the table. See Isolation levels and write conflicts on Azure Databricks.

Can I modify a Delta table from different workspaces?

Yes, you can concurrently modify the same Delta table from different workspaces. Moreover, if one process is writing from a workspace, readers in other workspaces will see a consistent view.

What is the medallion lakehouse architecture?

The medallion architecture describes a series of data layers that denote the quality of data stored in the lakehouse. Azure Databricks recommends taking a multi-layered approach to building a single source of truth for enterprise data products.

This architecture guarantees atomicity, consistency, isolation, and durability as data passes through multiple layers of validations and transformations before being stored in a layout optimized for efficient analytics. The terms bronze (raw), silver (validated), and gold (enriched) describe the quality of the data in each of these layers.

Medallion architecture as a data design pattern

A medallion architecture is a data design pattern used to organize data logically. Its goal is to incrementally and progressively improve the structure and quality of data as it flows through each layer of the architecture (from Bronze ⇒ Silver ⇒ Gold layer tables). Medallion architectures are sometimes also referred to as *multi-hop architectures*.

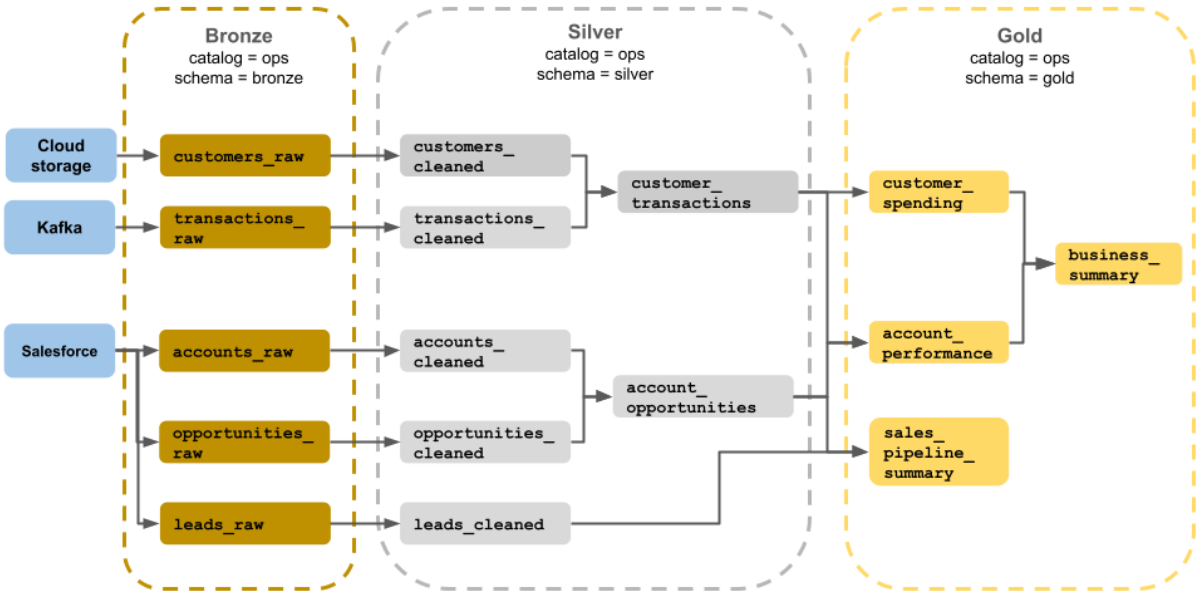
By progressing data through these layers, organizations can incrementally improve data quality and reliability, making it more suitable for business intelligence and machine learning applications.

Following the medallion architecture is a recommended best practice but not a requirement.

Question	Bronze	Silver	Gold
What happens in this layer?	Raw data ingestion	Data cleaning and validation	Dimensional modeling and aggregation
Who is the intended user?	• Data engineers • Data operations • Compliance and audit teams	• Data engineers • Data analysts (use the Silver layer for a more refined dataset that still retains detailed information necessary for in-depth analysis) • Data scientists (build models and perform advanced analytics)	• Business analysts and BI developers • Data scientists and machine learning (ML) engineers • Executives and decision makers • Operational teams

Example medallion architecture

This example of a medallion architecture shows bronze, silver, and gold layers for use by a business operations team. Each layer is stored in a different schema of the ops catalog.



- **Bronze layer** (ops.bronze): Ingests raw data from cloud storage, Kafka, and Salesforce. No data cleanup or validation is performed here.

- **Silver layer** (ops.silver): Data cleanup and validation are performed in this layer.
    - Data about customers and transactions is cleaned by dropping nulls and quarantining invalid records. These datasets are joined into a new dataset called customer_transactions. Data scientists can use this dataset for predictive analytics.
    - Similarly, accounts and opportunity datasets from Salesforce are joined to create account_opportunities, which is enhanced with account information.
    - The leads_raw data is cleaned in a dataset called leads_cleaned.
- **Gold layer** (ops.gold): This layer is designed for business users. It contains fewer datasets than silver and gold.
    - customer_spending: Average and total spend for each customer.
    - account_performance: Daily performance for each account.
    - sales_pipeline_summary: Information about the end-to-end sales pipeline.
    - business_summary: Highly aggregated information for the executive staff.

Ingest raw data to the bronze layer

The bronze layer contains raw, unvalidated data. Data ingested in the bronze layer typically has the following characteristics:

- Contains and maintains the raw state of the data source in its original formats.
- Is appended incrementally and grows over time.
- Is intended for consumption by workloads that enrich data for silver tables, not for access by analysts and data scientists.
- Serves as the single source of truth, preserving the data's fidelity.
- Enables reprocessing and auditing by retaining all historical data.
- Can be any combination of streaming and batch transactions from sources, including cloud object storage (for example, S3, GCS, ADLS), message buses (for example, Kafka, Kinesis, etc.), and federated systems (for example, Lakehouse Federation).

Limit data cleanup or validation

Minimal data validation is performed in the bronze layer. To ensure against dropped data, Azure Databricks recommends storing most fields as string, VARIANT, or binary to protect against unexpected schema changes. Metadata columns might be added, such as the provenance or source of the data (for example, _metadata.file_name).

Validate and deduplicate data in the silver layer

Data cleanup and validation are performed in silver layer.

Build silver tables from the bronze layer

To build the silver layer, read data from one or more bronze or silver tables, and write data to silver tables.

Azure Databricks does not recommend writing to silver tables directly from ingestion. If you write directly from ingestion, you'll introduce failures due to schema changes or corrupt records in data sources. Assuming all sources are append-only, configure most reads from bronze as streaming reads. Batch reads should be reserved for small datasets (for example, small dimensional tables).

The silver layer represents validated, cleaned, and enriched versions of the data. The silver layer:

- Should always include at least one validated, non-aggregated representation of each record. If aggregate representations drive many downstream workloads, those representations might be in the silver layer, but typically they are in the gold layer.
- Is where you perform data cleansing, deduplication, and normalization.
- Enhances data quality by correcting errors and inconsistencies.
- Structures data into a more consumable format for downstream processing.

Enforce data quality

The following operations are performed in silver tables:

- Schema enforcement
- Handling of null and missing values
- Data deduplication
- Resolution of out-of-order and late-arriving data issues
- Data quality checks and enforcement
- Schema evolution
- Type casting
- Joins

Start modeling data

It is common to start performing data modeling in the silver layer, including choosing how to represent heavily nested or semi-structured data:

- Use VARIANT data type.
- Use JSON strings.
- Create structs, maps, and arrays.
- Flatten schema or normalize data into multiple tables.

Power analytics with the gold layer

The gold layer represents highly refined views of the data that drive downstream analytics, dashboards, ML, and applications. Gold layer data is often highly aggregated and filtered for specific time periods or geographic regions. It contains semantically meaningful datasets that map to business functions and needs.

The gold layer:

- Consists of aggregated data tailored for analytics and reporting.
- Aligns with business logic and requirements.
- Is optimized for performance in queries and dashboards.

Align with business logic and requirements

The gold layer is where you'll model your data for reporting and analytics using a dimensional model by establishing relationships and defining measures. Analysts with access to data in gold should be able to find domain-specific data and answer questions.

Because the gold layer models a business domain, some customers create multiple gold layers to meet different business needs, such as HR, finance, and IT.

Create aggregates tailored for analytics and reporting

Organizations often need to create aggregate functions for measures like averages, counts, maximums, and minimums. For example, if your business needs to answer questions about total weekly sales, you could create a materialized view called weekly_sales that preaggregates this data so analysts and others don't need to recreate frequently used materialized views.

```
CREATE OR REPLACE MATERIALIZED VIEW weekly_sales AS
SELECT week,
    prod_id,
    region,
    SUM(units) AS total_units,
    SUM(units * rate) AS total_sales
FROM orders
GROUP BY week, prod_id, region
```

Optimize for performance in queries and dashboards

Optimizing gold-layer tables for performance is a best practice because these datasets are frequently queried. Large amounts of historical data are typically accessed in the sliver layer and not materialized in the gold layer.

Control costs by adjusting the frequency of data ingestion

Control costs by determining how frequently to ingest data.

Data ingestion frequency	**Cost**	**Latency**	**Declarative examples**	**Procedural examples**
Continuous incremental ingestion	Higher	Lower	• Streaming Table using spark.readStream to ingest from cloud storage or message bus.	•

Data ingestion frequency	Cost	Latency	Declarative examples	Procedural examples
			• The pipeline that updates this streaming table runs continuously. • Structured Streaming code using spark.readStream in a notebook to ingest from cloud storage or message bus into a Delta table. • The notebook is orchestrated using an Azure Databricks job with a continuous job trigger.	
Triggered incremental ingestion	Lower	Higher	• Streaming Table ingesting from cloud storage or message bus using spark.readStream. • The pipeline that updates this streaming table is triggered by the job's scheduled trigger or a file arrival trigger. • Structured Streaming code in a notebook with a Trigger.Available trigger. • This notebook is triggered by the job's scheduled trigger or a file arrival trigger.	•
Batch ingestion with manual incremental ingestion	Lower	Highest, because of infrequent runs.	• Streaming Table ingest from cloud storage using spark.read. • Does not use Structured Streaming. Instead, use primitives like partition overwrite to update an entire partition at one time. • Requires extensive upstream architecture to set up the incremental processing, which allows for a cost similar to Structured Streaming reads/writes. • Also requires partitioning source data by a datetime field and then	

Data ingestion frequency	Cost	Latency	Declarative examples	Procedural examples
			processing all records from that partition into the target.	

What does it mean to build a single source of truth?

The Databricks lakehouse eliminates the need for creating and syncing copies of data across multiple systems by unifying data access and storage in a single system, establishing the lakehouse as the single source of truth (SSOT). Duplicating data often results in data silos, meaning that different teams within an organization may be working with versions of the same data that differ in quality and freshness.

How does the lakehouse control transactions and data access?

Delta Lake transactions use log files stored alongside data files to provide ACID guarantees at a table level. Because the data and log files backing Delta Lake tables live together in cloud object storage, reading and writing data can occur simultaneously without risk of many queries resulting in performance degradation or deadlock for business-critical workloads. This means that users and applications throughout the enterprise environment can connect to the same single copy of the data to drive diverse workloads, with all viewers guaranteed to receive the most current version of the data at the time their query executes.

Manage access to production data

Unity Catalog provides a centralized data governance solution that allows data stewards to provide fine-grained access control to users, groups, and service principals. Unity Catalog governs permissions using access control lists (ACLs) that provide both flexibility and specificity in configuring resources. Some configurable permissions include:

- Read-only access to a handful of tables.
- Table creation and modification permissions for a database.
- Ability to read or modify data in a specific cloud storage location.
- Access to many cloud resources through Unity Catalog managed storage credentials.

For more information, see What is Unity Catalog?.

Leverage views in the lakehouse

Views on Azure Databricks represent saved queries against data stored in tables somewhere in the lakehouse. Whereas the queries that result in tables are executed at write time, views execute defining logic each time a query against a view runs. This means that views can provide up-to-date access to data from a variety of sources, and that compute is only spent to update results as they are needed.

You can use Unity Catalog to secure and share views alongside other data objects, allowing individuals and teams to share the logic that drives key business decisions across the organization.

For more information, see What is a view?.

Share data with collaborators

While the ACLs in Unity Catalog cover a wide range of use cases for sharing data within an enterprise organization, Delta Sharing further expands this by managing read-only access to datasets that can be shared with collaborators anywhere. Use cases supported by Unity Catalog include:

- Providing real-time access to regional analytics for isolated regions of multinational corporations.
- Sharing datasets across isolated businesses that exist under the same corporate umbrella.
- Providing secure access to customer-curated datasets for third-party consumers.

On Azure Databricks, Delta Sharing comes built-in with Unity Catalog, but it is also part of open source Delta Lake.

Data discovery and collaboration in the lakehouse

Databricks designed Unity Catalog to help organizations reduce time to insights by empowering a broader set of data users to discover and analyze data at scale. Data stewards can securely grant access to data assets for diverse teams of end users in Unity Catalog. These users can then use a variety of languages and tools, including SQL and Python, to create derivative datasets, models, and dashboards that can be shared across teams.

Manage permissions at scale

Unity Catalog provides administrators a unified location to assign permissions for catalogs, databases, tables, and views to groups of users. Privileges and metastores are shared across workspaces, allowing administrators to set secure permissions once against groups synced from identity providers and know that end users only have access to the proper data in any Azure Databricks workspace they enter.

Unity Catalog also allows administrators to define storage credentials, a secure way to store and share permissions on cloud storage infrastructure. You can grant privileges on these securables to power users within the organization so they can define external locations against cloud object storage locations, allowing data engineers to self-service for new workloads without needing to provide elevated permissions in cloud account consoles.

Discover data on Azure Databricks

Users can browse available data objects in Unity Catalog using Catalog Explorer. Catalog Explorer uses the privileges configured by Unity Catalog administrators to ensure that users are only able to see catalogs, databases, tables, and views that they have permissions to query. Once users find a dataset of interest, they can review field names and types, read comments on tables and individual fields, and preview a sample of the data. Users can also review the full history of the table to understand when and how data has changed, and the lineage feature allows users to track how certain datasets are derived from upstream jobs and used in downstream jobs.

Storage credentials and external locations are also displayed in Catalog Explorer, allowing each user to fully grasp the privileges they have to read and write data across available locations and resources.

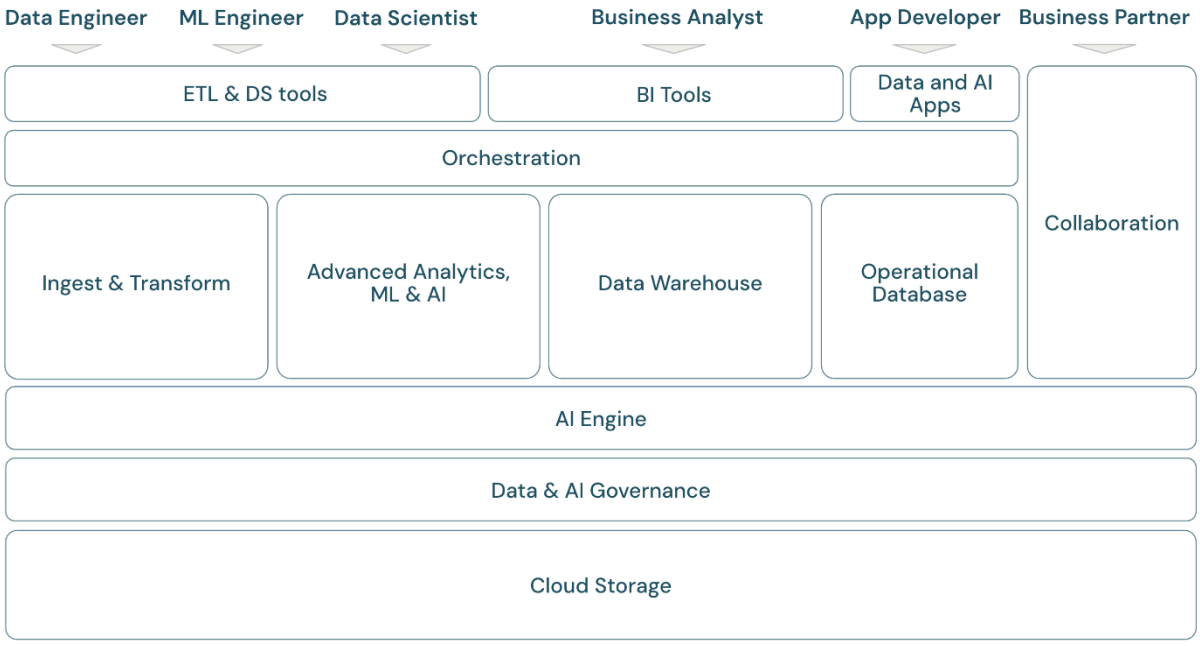
Accelerate time to production with the lakehouse

Azure Databricks supports workloads in SQL, Python, Scala, and R, allowing users with diverse skill sets and technical backgrounds to leverage their knowledge to derive analytic insights. You can use all languages supported by Azure Databricks to define production jobs, and notebooks can leverage a combination of languages. This means that you can promote queries written by SQL analysts for last mile ETL into production data engineering code with almost no effort. Queries and workloads defined by personas across the organization leverage the same datasets, so there's no need to reconcile field names or make sure dashboards are up to date before sharing code and results with other teams. You can securely share code, notebooks, queries, and dashboards, all powered by the same scalable cloud infrastructure and defined against the same curated data sources.

The scope of the lakehouse platform

A modern data and AI platform framework

To discuss the scope of the Databricks Data intelligence Platform, it is helpful to first define a basic framework for the modern data and AI platform:



Overview of the lakehouse scope

The Databricks Data Intelligence Platform covers the complete modern data platform framework. It is built on the lakehouse architecture and powered by a data intelligence engine that understands the unique qualities of your data. It is an open and unified foundation for ETL, ML/AI, and DWH/BI workloads, and has Unity Catalog as the central data and AI governance solution.

Personas of the platform framework

The framework covers the primary data team members (personas) working with the applications in the framework:

- **Data engineers** provide data scientists and business analysts with accurate and reproducible data for timely decision-making and real-time insights. They implement highly consistent and reliable ETL processes to increase user confidence and trust in data. They ensure that data is well integrated with the various pillars of the business and typically follow software engineering best practices.
- **Data scientists** blend analytical expertise and business understanding to transform data into strategic insights and predictive models. They are adept at translating business challenges into data-driven solutions, be that through retrospective analytical insights or forward-looking predictive modeling. Leveraging data modeling and machine learning techniques, they design, develop, and deploy models that unveil patterns, trends, and forecasts from data. They act as a bridge, converting complex data narratives into comprehensible stories, ensuring business stakeholders not only understand but can also act upon the data-driven recommendations, in turn driving a data-centric approach to problem-solving within an organization.
- **ML engineers** (machine learning engineers) lead the practical application of data science in products and solutions by building, deploying, and maintaining machine learning models. Their primary focus pivots towards the engineering aspect of model development and deployment. ML Engineers ensure the robustness, reliability, and scalability of machine learning systems in live environments, addressing challenges related to data quality, infrastructure, and performance. By integrating AI and ML models into operational business processes and user-facing products, they facilitate the utilization of data science in solving business challenges, ensuring models don't just stay in research but drive tangible business value.
- **Business analysts** and **business users**: Business analysts provide stakeholders and business teams with actionable data. They often interpret data and create reports or other documentation for management using standard BI tools. They are typically the first point of contact for non-technical business users and operations colleagues for quick analysis questions. Dashboards and business apps delivered on the Databricks platform can be used directly by business users.
- **Apps Developer** create secure data and AI applications on the data platform and share those apps with business users.
- **Business partners** are important stakeholders in an increasingly networked business world. They are defined as a company or individuals with whom a business has a formal relationship to achieve a common goal, and can include vendors, suppliers, distributors, and other third-party partners. Data sharing is an important aspect of business partnerships, as it enables the transfer and exchange of data to enhance collaboration and data-driven decision-making.

Domains of the platform framework

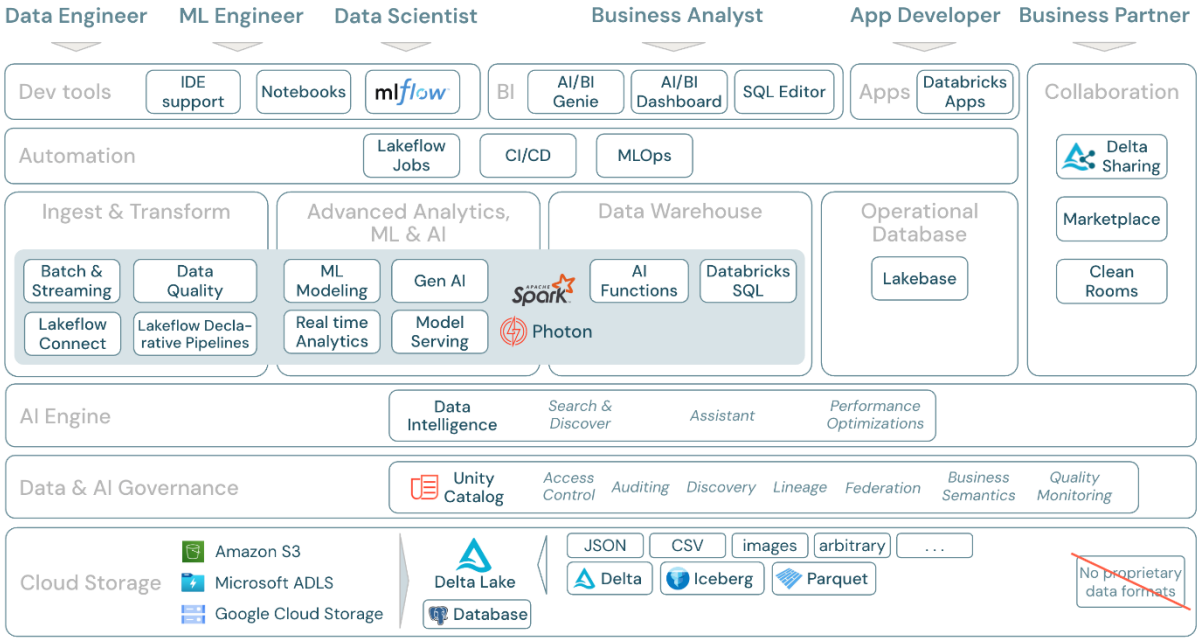
The platform consists of multiple domains:

- **Storage:** In the cloud, data is mainly stored in scalable, efficient, and resilient object storage on cloud providers.

- **Governance:** Capabilities around data governance, such as access control, auditing, metadata management, lineage tracking, and monitoring for all data and AI assets.
- **AI engine:** The AI engine provides generative AI capabilities for the whole platform.
- **Ingest and transform:** The capabilities for ETL workloads.
- **Advanced analytics, ML, and AI:** All capabilities around machine learning, AI, Generative AI, and also streaming analytics.
- **Data warehouse:** The domain supporting DWH and BI use cases.
- **Operational Database:** Capabilities and services around operational databases like OLTP databases (online transaction processing), key-value stores, etc.
- **Automation:** Workflow management for data processing, machine learning, analytics pipelines, including CI/CD and MLOps support.
- **ETL and Data science tools:** The front-end tools that data engineers, data scientists and ML engineers primarily use for work.
- **BI tools:** The front-end tools that BI analysts primarily use for work.
- **Data and AI apps** Tools that build and host applications that use the data managed by the underlying platform and leverage its analytics and AI capabilities in a secure and governance-compliant manner.
- **Collaboration**: Capabilities for data sharing between two or more parties.

The scope of the Databricks Platform

The Databricks Data Intelligence Platform and its components can be mapped to the framework in the following way:



[Download: Scope of the lakehouse - Databricks components]

Data workloads on Azure Databricks

Most importantly, the Databricks Data Intelligence Platform covers all relevant workloads for the data domain in one platform, with Apache Spark/Photon as the engine:

- **Ingest and transform**

Databricks offers several ways of data ingestion:

  - Databricks Lakeflow Connect offers built-in connectors for ingestion from enterprise applications and databases. The resulting ingestion pipeline is governed by Unity Catalog and is powered by serverless compute and Lakeflow Declarative Pipelines.
  - Auto Loader incrementally and automatically processes files landing in cloud storage in scheduled or continuous jobs - without the need to manage state information. Once ingested, raw data needs to be transformed so it's ready for BI and ML/AI. Databricks provides powerful ETL capabilities for data engineers, data scientists, and analysts.

Lakeflow Declarative Pipelines allows writing ETL jobs in a declarative way, simplifying the entire implementation process. Data quality can be improved by defining data expectations.

- **Advanced analytics, ML, and AI**

The platform includes *Databricks Mosaic AI*, a set of fully integrated machine learning and AI tools for traditional machine and deep learning, as well as generative AI and large language models (LLMs). It covers the entire workflow from preparing data to building machine learning and deep learning models, to Mosaic AI Model Serving.

Spark Structured Streaming and Lakeflow Declarative Pipelines enable real-time analytics.

- **Data warehouse**

The Databricks Data Intelligence Platform also has a complete data warehouse solution with Databricks SQL, centrally governed by Unity Catalog with fine-grained access control.

AI functions are built-in SQL functions that allow you to apply AI on your data directly from SQL. Integrating AI into analysis jobs provides access to information previously inaccessible to analysts, and empowers them to make more informed decisions, manage risks, and sustain a competitive advantage through data-driven innovation and efficiency.

- **Operational database**

Lakebase is an online transaction processing (OLTP) database based on Postgres and fully integrated with the Databricks Data Intelligence Platform. It allows you to create an OLTP database on Databricks, and integrate OLTP workloads with your Lakehouse. Lakebase allows to sync data between OLTP and online analytical processing (OLAP) workloads, and is well integrated with Feature management, SQL warehouses, and Databricks Apps.

Outline of Azure Databricks feature areas

This is a mapping of the Databricks Data Intelligence Platform features to the other layers of the framework, from bottom to top:

- **Cloud storage**

All data for the lakehouse is stored in the cloud provider's object storage. Databricks supports three cloud providers: AWS, Azure, and GCP. Files in various structured and semi-structured formats (for example, Parquet, CSV, JSON, and Avro), as well as unstructured formats (such as images and documents), are ingested and transformed using either batch or streaming processes.

Delta Lake is the recommended data format for the lakehouse (file transactions, reliability, consistency, updates, and so on). It's also possible to read Delta tables using Apache Iceberg clients.

No proprietary data formats are used in the Databricks Data Intelligence Platform: Delta Lake and Iceberg are open source to avoid vendor lock-in.

- **Data and AI governance**

On top of the storage layer, Unity Catalog offers a wide range of data and AI governance capabilities, including metadata management in the metastore, access control, auditing, data discovery, and data lineage.

Lakehouse monitoring provides out-of-the-box quality metrics for data and AI assets, and auto-generated dashboards to visualize these metrics.

External SQL sources can be integrated into the lakehouse and Unity Catalog through lakehouse federation.

- **AI engine**

The Data Intelligence Platform is built on the lakehouse architecture and enhanced by Databricks AI-powered features. Databricks AI combines generative AI with the unification benefits of the lakehouse architecture to understand the unique semantics of your data. Intelligent Search and the Databricks Assistant are examples of AI powered services that simplify working with the platform for every user.

- **Orchestration**

Lakeflow Jobs enable you to run diverse workloads for the full data and AI lifecycle on any cloud. They allow you to orchestrate jobs as well as Lakeflow Declarative Pipelines for SQL, Spark, notebooks, DBT, ML models, and more.

The platform also supports CI/CD and MLOps

- **ETL & DS tools**

At the consumption layer, data engineers and ML engineers typically work with the platform using IDEs. Data scientists often prefer notebooks and use the ML & AI runtimes, and the machine learning workflow system MLflow to track experiments and manage the model lifecycle.

- **BI tools**

Business analysts typically use their preferred BI tool to access the Databricks data warehouse. Databricks SQL can be queried by different Analysis and BI tools, see BI and visualization

In addition, the platform offers query and analysis tools out of the box:

- AI/BI Dashboards to drag-and-drop data visualizations and share insights.
- Domain experts, such as data analysts, configure AI/BI Genie spaces with datasets, sample queries, and text guidelines to help Genie translate business questions into analytical queries. After set up, business users can ask questions and generate visualizations to understand operational data.
- SQL editor for SQL analysts to analyze data.

- **Data and AI apps**

Databricks Apps lets developers create secure data and AI applications on the Databricks platform and share those apps with users.

- **Collaboration**

Delta Sharing is an open protocol developed by Databricks for secure data sharing with other organizations regardless of the computing platforms they use.

Databricks Marketplace is an open forum for exchanging data products. It takes advantage of Delta Sharing to give data providers the tools to share data products securely and data consumers the power to explore and expand their access to the data and data services they need.

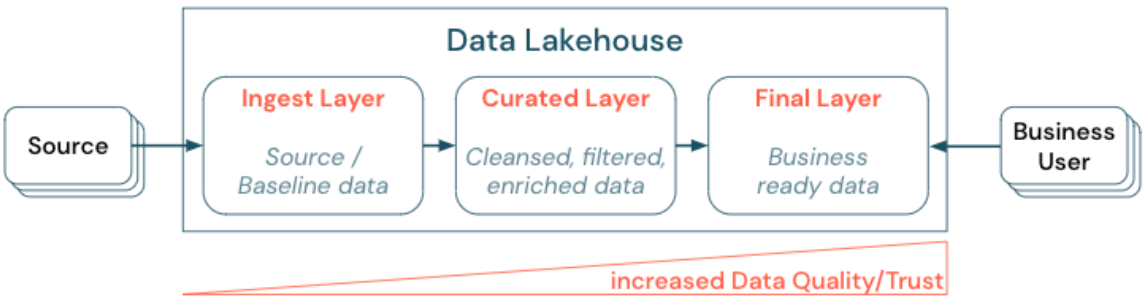
Clean Rooms use Delta Sharing and serverless compute to provide a secure and privacy-protecting environment where multiple parties can work together on sensitive enterprise data without direct access to each other's data.

Guiding principles for the lakehouse

Guiding principles are level-zero rules that define and influence your architecture. To build a data lakehouse that helps your business succeed now and in the future, consensus among stakeholders in your organization is critical.

Curate data and offer trusted data-as-products

Curating data is essential to creating a high-value data lake for BI and ML/AI. Treat data like a product with a clear definition, schema, and lifecycle. Ensure semantic consistency and that the data quality improves from layer to layer so that business users can fully trust the data.



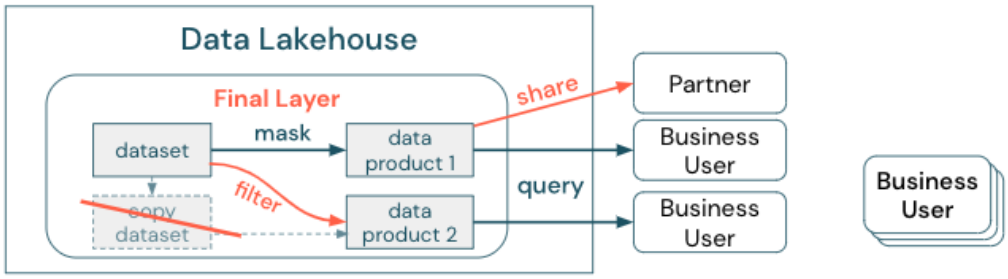
Curating data by establishing a layered (or multi-hop) architecture is a critical best practice for the lakehouse, as it allows data teams to structure the data according to quality levels and define roles and responsibilities per layer. A common layering approach is:

- **Ingest layer**: Source data gets ingested into the lakehouse into the first layer and should be persisted there. When all downstream data is created from the ingest layer, rebuilding the subsequent layers from this layer is possible, if needed.
- **Curated layer**: The purpose of the second layer is to hold cleansed, refined, filtered and aggregated data. The goal of this layer is to provide a sound, reliable foundation for analyses and reports across all roles and functions.
- **Final layer**: The third layer is created around business or project needs; it provides a different view as data products to other business units or projects, preparing data around security needs (for example, anonymized data), or optimizing for performance (with pre-aggregated views). The data products in this layer are seen as the truth for the business.

Pipelines across all layers need to ensure that data quality constraints are met, meaning that data is accurate, complete, accessible, and consistent at all times, even during concurrent reads and writes. The validation of new data happens at the time of data entry into the curated layer, and the following ETL steps work to improve the quality of this data. Data quality must improve as data progresses through the layers and, as such, the trust in the data subsequently increases from a business point of view.

Eliminate data silos and minimize data movement

Don't create copies of a dataset with business processes relying on these different copies. Copies may become data silos that get out of sync, leading to lower quality of your data lake, and finally to outdated or incorrect insights. Also, for sharing data with external partners, use an enterprise sharing mechanism that allows direct access to the data in a secure way.



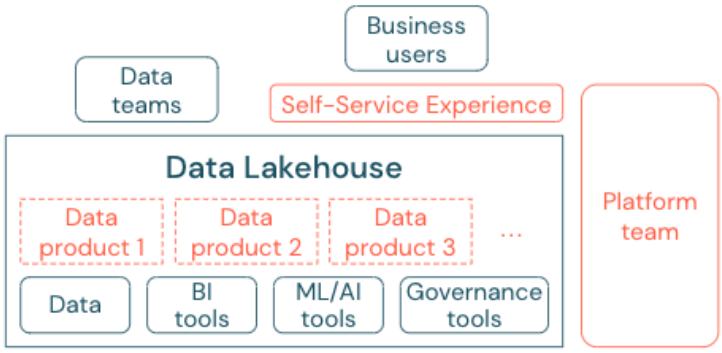
To make the distinction clear between a data copy versus a data silo: A standalone or throwaway copy of data is not harmful on its own. It is sometimes necessary for boosting agility, experimentation, and innovation. However, if these copies become operational with downstream business data products dependent on them, they become data silos.

To prevent data silos, data teams usually attempt to build a mechanism or data pipeline to keep all copies in sync with the original. Since this is unlikely to happen consistently, data quality eventually degrades. This can also lead to higher costs and a significant loss of user trust. On the other hand, several business use cases require data sharing with partners or suppliers.

An important aspect is to securely and reliably share the latest version of the dataset. Copies of the dataset are often not sufficient, because they can get out of sync quickly. Instead, data should be shared via enterprise data-sharing tools.

Democratize value creation through self-service

The best data lake cannot provide sufficient value, if users cannot access the platform or data for their BI and ML/AI tasks easily. Lower the barriers to accessing data and platforms for all business units. Consider lean data management processes and provide self-service access for the platform and the underlying data.



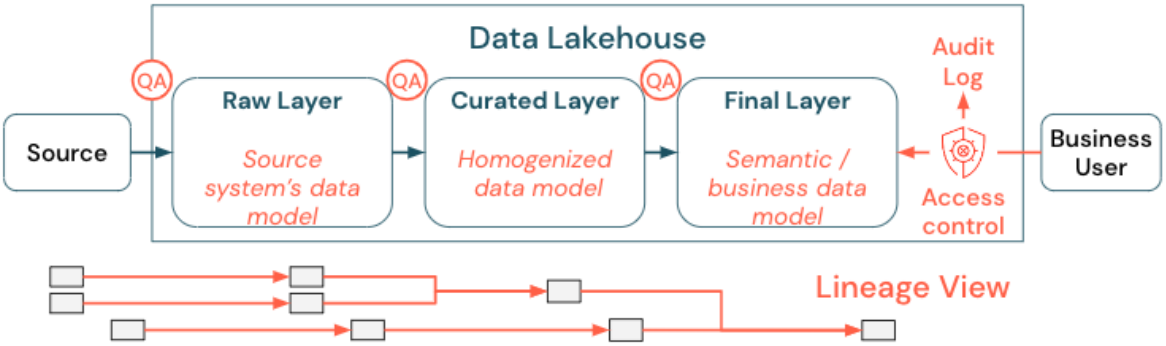
Businesses that have successfully moved to a data-driven culture will thrive. This means every business unit derives its decisions from analytical models or from analyzing its own or centrally provided data. For consumers, data has to be easily discoverable and securely accessible.

A good concept for data producers is “data as a product”: The data is offered and maintained by one business unit or business partner like a product and consumed by other parties with proper permission control. Instead of relying on a central team and potentially slow request processes, these data products must be created, offered, discovered, and consumed in a self-service experience.

However, it's not just the data that matters. The democratization of data requires the right tools to enable everyone to produce or consume and understand the data. For this, you need the data lakehouse to be a modern data and AI platform that provides the infrastructure and tooling for building data products without duplicating the effort of setting up another tool stack.

Adopt an organization-wide data and AI governance strategy

Data is a critical asset of any organization, but you cannot give everyone access to all data. Data access must be actively managed. Access control, auditing, and lineage-tracking are key for the correct and secure use of data.



Data governance is a broad topic. The lakehouse covers the following dimensions:

- **Data quality**

The most important prerequisite for correct and meaningful reports, analysis results, and models is high-quality data. Quality assurance (QA) needs to exist around all pipeline steps. Examples of how to implement this include having data contracts, meeting SLAs, keeping schemas stable, and evolving them in a controlled way.

- **Data catalog**

Another important aspect is data discovery: Users of all business areas, especially in a self-service model, must be able to discover relevant data easily. Therefore, a lakehouse needs a data catalog that covers all business-relevant data. The primary goals of a data catalog are as follows:

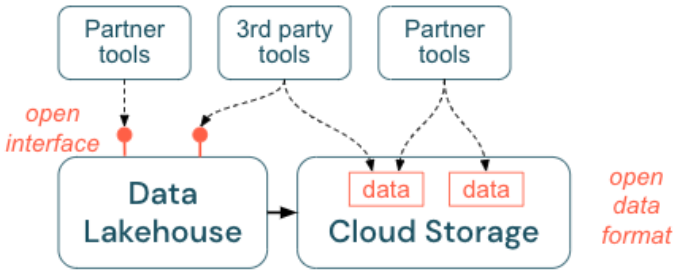
  - Ensure the same business concept is uniformly called and declared across the business. You might think of it as a semantic model in the curated and the final layer.
  - Track the data lineage precisely so that users can explain how these data arrived at their current shape and form.
  - Maintain high-quality metadata, which is as important as the data itself for proper use of the data.

- **Access control**

As the value creation from the data in the lakehouse happens across all business areas, the lakehouse must be built with security as a first-class citizen. Companies might have a more open data access policy or strictly follow the principle of least privileges. Independent of that, data access controls must be in place in every layer. It is important to implement fine-grade permission schemes from the very beginning (column- and row-level access control, role-based or attribute-based access control). Companies can start with less strict rules. But as the lakehouse platform grows, all mechanisms and processes for a more sophisticated security regime should already be in place. Additionally, all access to the data in the lakehouse must be governed by audit logs from the get-go.

Encourage open interfaces and open formats

Open interfaces and data formats are crucial for interoperability between the lakehouse and other tools. It simplifies integration with existing systems and also opens up an ecosystem of partners who have integrated their tools with the platform.



Open interfaces are critical to enabling interoperability and preventing dependency on any single vendor. Traditionally, vendors built proprietary technologies and closed interfaces that limited enterprises in the way they can store, process and share data.

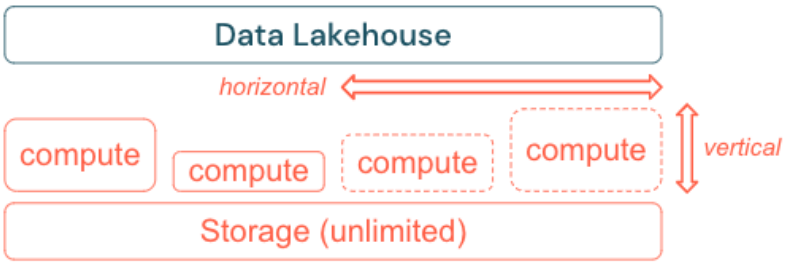
Building upon open interfaces helps you build for the future:

- It increases the longevity and portability of the data so that you can use it with more applications and for more use cases.
- It opens an ecosystem of partners who can quickly leverage the open interfaces to integrate their tools into the lakehouse platform.

Finally, by standardizing on open formats for data, total costs will be significantly lower; one can access the data directly on the cloud storage without the need to pipe it through a proprietary platform that can incur high egress and computation costs.

Build to scale and optimize for performance and cost

Data inevitably continues to grow and become more complex. To equip your organization for future needs, your lakehouse should be able to scale. For example, you should be able to add new resources easily on demand. Costs should be limited to the actual consumption.



Standard ETL processes, business reports, and dashboards often have a predictable resource need from a memory and computation perspective. However, new projects, seasonal tasks, or modern approaches like model training (churn, forecast, maintenance) generate peaks of resource need. To enable a business to perform all these workloads, a scalable platform for memory and computation is necessary. New resources must be added easily on demand, and only the actual consumption should generate costs. As soon as the peak is over, resources can be freed up again and costs reduced accordingly. Often, this is referred to as horizontal scaling (fewer or more nodes) and vertical scaling (larger or smaller nodes).

Scaling also enables businesses to improve the performance of queries by selecting nodes with more resources or clusters with more nodes. But instead of permanently providing large machines and clusters they can be provisioned on demand only for the time needed to optimize the overall performance to cost ratio. Another aspect of optimization is storage versus compute resources. Since there is no clear relation between the volume of the data and workloads using this data (for example, only using parts of the data or doing intensive calculations on small data), it is a good practice to settle on an infrastructure platform that decouples storage and compute resources.

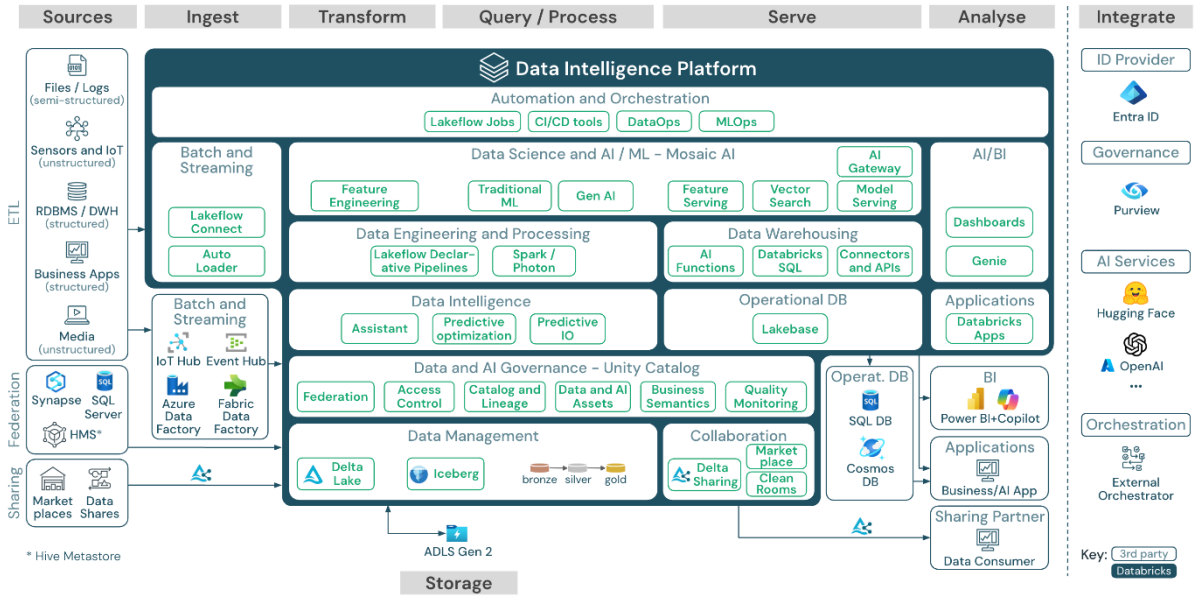
Lakehouse reference architectures (download)

This article provides architectural guidance for the lakehouse, covering data sources, ingestion, transformation, querying and processing, serving, analysis, and storage.

Each reference architecture has a downloadable PDF in 11 x 17 (A3) format.

While the lakehouse on Databricks is an open platform that integrates with a large ecosystem of partner tools, the reference architectures focus only on Azure services and the Databricks

lakehouse. The cloud provider services shown are selected to illustrate the concepts and are not exhaustive.



[Download: Reference architecture for the Azure Databricks lakehouse]

The Azure reference architecture shows the following Azure-specific services for ingesting, storage, serving, and analysis:

- Azure Synapse and SQL Server as source systems for Lakehouse Federation
- Azure IoT Hub and Azure Event Hubs for streaming ingest
- Azure Data Factory for batch ingest
- Azure Data Lake Storage Gen 2 (ADLS) as the object storage for data and AI assets
- Azure SQL DB and Azure Cosmos DB as operational databases
- Azure Purview as the enterprise catalog to which UC exports schema and lineage information
- Power BI as the BI tool
- Azure OpenAI can be used by Model Serving as an external LLM

Organization of the reference architectures

The reference architecture is structured along the swim lanes *Source*, *Ingest*, *Transform*, *Query/Process*, *Serve*, *Analysis*, and *Storage*:

- **Source**

There are three ways to integrate external data into the Data Intelligence Platform:

  - ETL: The platform enables integration with systems that provide semi-structured and unstructured data (such as sensors, IoT devices, media, files, and logs), as well as structured data from relational databases or business applications.

- Lakehouse Federation: SQL sources, such as relational databases, can be integrated into the lakehouse and Unity Catalog without ETL. In this case, the source system data is governed by Unity Catalog, and queries are pushed down to the source system.
- Catalog Federation: Hive Metastore catalogs can also be integrated into Unity Catalog through catalog federation, allowing Unity Catalog to control the tables stored in Hive Metastore.

- **Ingest**

Ingest data into the lakehouse via batch or streaming:

- Databricks Lakeflow Connect offers built-in connectors for ingestion from enterprise applications and databases. The resulting ingestion pipeline is governed by Unity Catalog and is powered by serverless compute and Lakeflow Declarative Pipelines.
- Files delivered to cloud storage can be loaded directly using the Databricks Auto Loader.
- For batch ingestion of data from enterprise applications into Delta Lake, the Databricks lakehouse relies on partner ingest tools with specific adapters for these systems of record.
- Streaming events can be ingested directly from event streaming systems such as Kafka using Databricks Structured Streaming. Streaming sources can be sensors, IoT, or change data capture processes.

- **Storage**
  - Data is typically stored in the cloud storage system where the ETL pipelines use the medallion architecture to store data in a curated way as Delta files/tables or Apache Iceberg tables.

- **Transform** and **Query / process**
  - The Databricks lakehouse uses its engines Apache Spark and Photon for all transformations and queries.
  - Lakeflow Declarative Pipelines is a declarative framework for simplifying and optimizing reliable, maintainable, and testable data processing pipelines.
  - Powered by Apache Spark and Photon, the Databricks Data Intelligence Platform supports both types of workloads: SQL queries via SQL warehouses, and SQL, Python and Scala workloads via workspace clusters.
  - For data science (ML Modeling and Gen AI), the Databricks AI and Machine Learning platform provides specialized ML runtimes for AutoML and for coding ML jobs. All data science and MLOps workflows are best supported by MLflow.

- **Serving**

- For data warehousing (DWH) and BI use cases, the Databricks lakehouse provides Databricks SQL, the data warehouse powered by SQL warehouses, and serverless SQL warehouses.
- For machine learning, Mosaic AI Model Serving is a scalable, real-time, enterprise-grade model serving capability hosted in the Databricks control plane. Mosaic AI Gateway is Databricks solution for governing and monitoring access to supported generative AI models and their associated model serving endpoints.
- Operational databases:
  - Lakebase is an online transaction processing (OLTP) database based on Postgres and fully integrated with the Databricks Data Intelligence Platform. It allows you to create OLTP databases on Databricks, and integrate OLTP workloads with your Lakehouse.
  - External systems, such as operational databases, can be used to store and deliver final data products to user applications.

- **Collaboration**:
  - Business partners get secure access to the data they need through Delta Sharing.
  - Based on Delta Sharing, the Databricks Marketplace is an open forum for exchanging data products.
  - Clean Rooms are secure and privacy-protecting environments where multiple users can work together on sensitive enterprise data without direct access to each other's data.
- **Analysis**
  - The final business applications are in this swim lane. Examples include custom clients such as AI applications connected to Mosaic AI Model Serving for real-time inference or applications that access data pushed from the lakehouse to an operational database.
  - For BI use cases, analysts typically use BI tools to access the data warehouse. SQL developers can additionally use the Databricks SQL Editor (not shown in the diagram) for queries and dashboarding.
  - The Data Intelligence Platform also offers dashboards to build data visualizations and share insights.
- **Integrate**
  - The Databricks platform integrates with standard identity providers for user management and single sign on (SSO).
  - External AI services like OpenAI, LangChain or HuggingFace can be used directly from within the Databricks Intelligence Platform.

- External orchestrators can either use the comprehensive REST API or dedicated connectors to external orchestration tools like Apache Airflow.
- Unity Catalog is used for all data & AI governance in the Databricks Intelligence Platform and can integrate other databases into its governance through Lakehouse Federation.

Additionally, Unity Catalog can be integrated into other enterprise catalogs, e.g. Purview. Contact the enterprise catalog vendor for details.

Common capabilities for all workloads

In addition, the Databricks lakehouse comes with management capabilities that support all workloads:

- **Data and AI governance**

The central data and AI governance system in the Databricks Data Intelligence Platform is Unity Catalog. Unity Catalog provides a single place to manage data access policies that apply across all workspaces and supports all assets created or used in the lakehouse, such as tables, volumes, features (feature store), and models (model registry). Unity Catalog can also be used to capture runtime data lineage across queries run on Databricks.

Databricks lakehouse monitoring allows you to monitor the data quality of all of the tables in your account. It can also track the performance of machine learning models and model-serving endpoints.

For observability, system tables is a Databricks-hosted analytical store of your account's operational data. System tables can be used for historical observability across your account.

- **Data intelligence engine**

The Databricks Data Intelligence Platform allows your entire organization to use data and AI, combining generative AI with the unification benefits of a lakehouse to understand the unique semantics of your data. See Databricks AI-powered features.

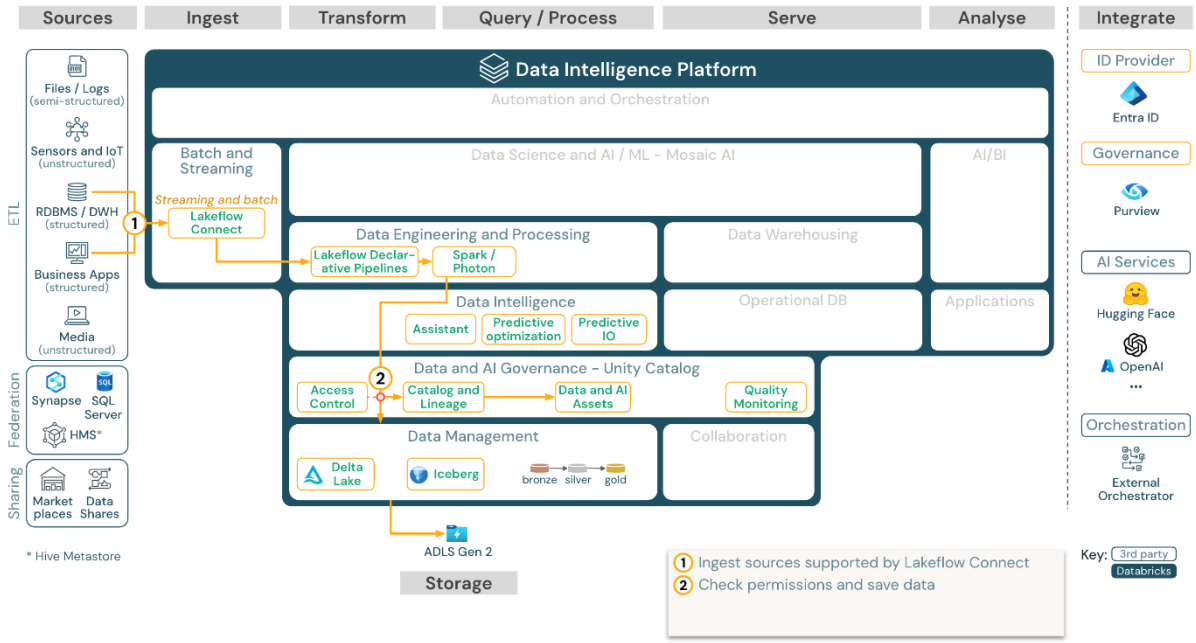
The Databricks Assistant is available in Databricks notebooks, SQL editor, file editor, and elsewhere as a context-aware AI assistant for users.

- **Automation & Orchestration**

Lakeflow Jobs orchestrate data processing, machine learning, and analytics pipelines on the Databricks Data Intelligence Platform. Lakeflow Declarative Pipelines allow you to build reliable and maintainable ETL pipelines with declarative syntax. The platform also supports CI/CD and MLOps

High-level use cases for the Data Intelligence Platform on Azure

Built-in ingestion from SaaS apps and databases with Lakeflow Connect

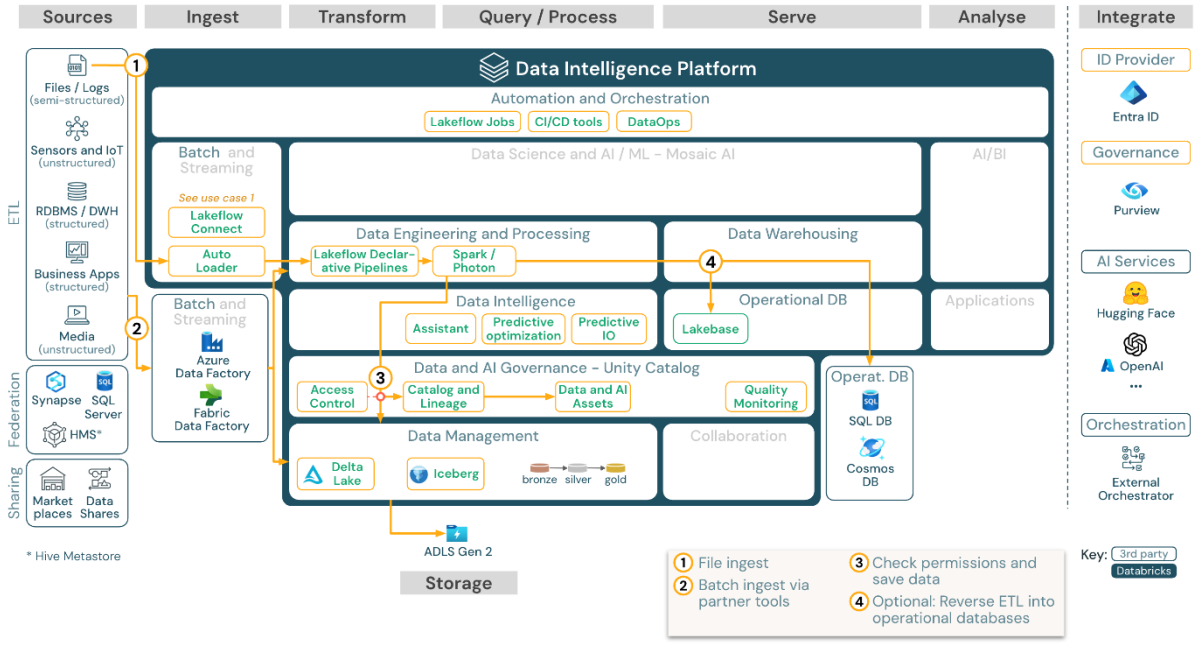


[Download: Lakeflow Connect reference architecture for Azure Databricks.]

Databricks Lakeflow Connect offers built-in connectors for ingestion from enterprise applications and databases. The resulting ingestion pipeline is governed by Unity Catalog and is powered by serverless compute and Lakeflow Declarative Pipelines.

Lakeflow Connect leverages efficient incremental reads and writes to make data ingestion faster, scalable, and more cost-efficient, while your data remains fresh for downstream consumption.

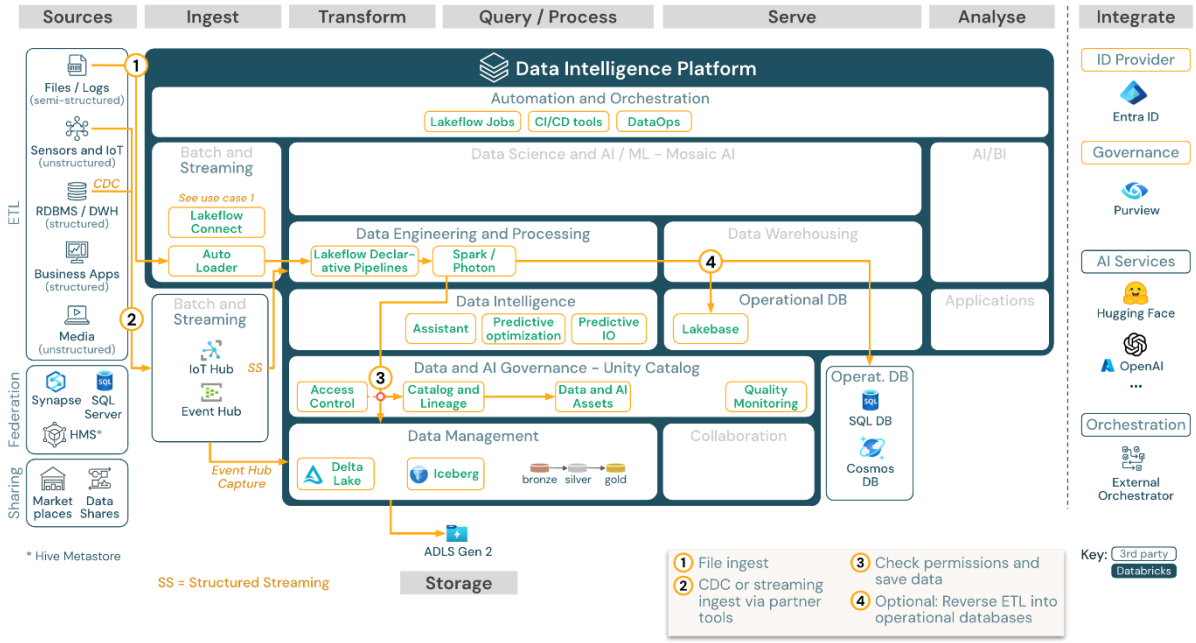
Batch ingestion and ETL



[Download: Batch ETL reference architecture for Azure Databricks]

Ingestion tools use source-specific adapters to read data from the source and then either store it in the cloud storage from where Auto Loader can read it, or call Databricks directly (for example, with partner ingestion tools integrated into the Databricks lakehouse). To load the data, the Databricks ETL and processing engine runs the queries via Lakeflow Declarative Pipelines. Orchestrate single or multitask jobs using Lakeflow Jobs and govern them using Unity Catalog (access control, audit, lineage, and so on). To provide access to specific golden tables for low-latency operational systems, export the tables to an operational database such as an RDBMS or key-value store at the end of the ETL pipeline.

Streaming and change data capture (CDC)



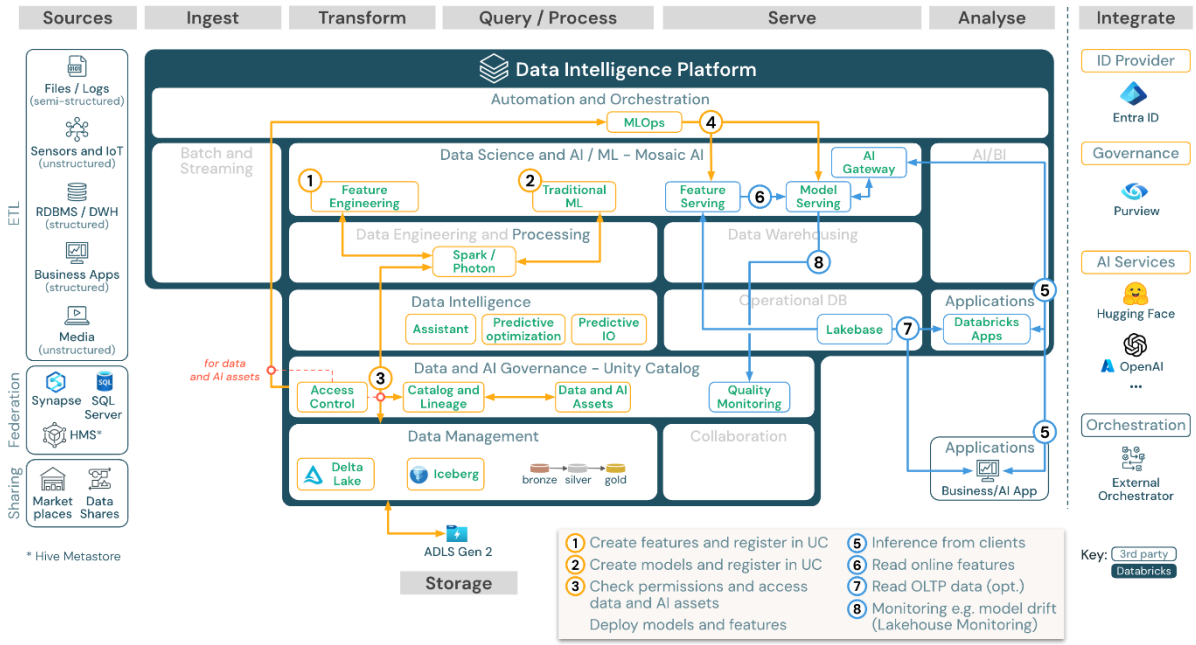
[Download: Spark structured streaming architecture for Azure Databricks]

The Databricks ETL engine uses Spark Structured Streaming to read from event queues such as Apache Kafka or Azure Event Hub. The downstream steps follow the approach of the Batch use case above.

Real-time change data capture (CDC) typically uses an event queue to store the extracted events. From there, the use case follows the streaming use case.

If CDC is done in batch where the extracted records are stored in cloud storage first, then Databricks Autoloader can read them and the use case follows Batch ETL.

Machine learning and AI (traditional)



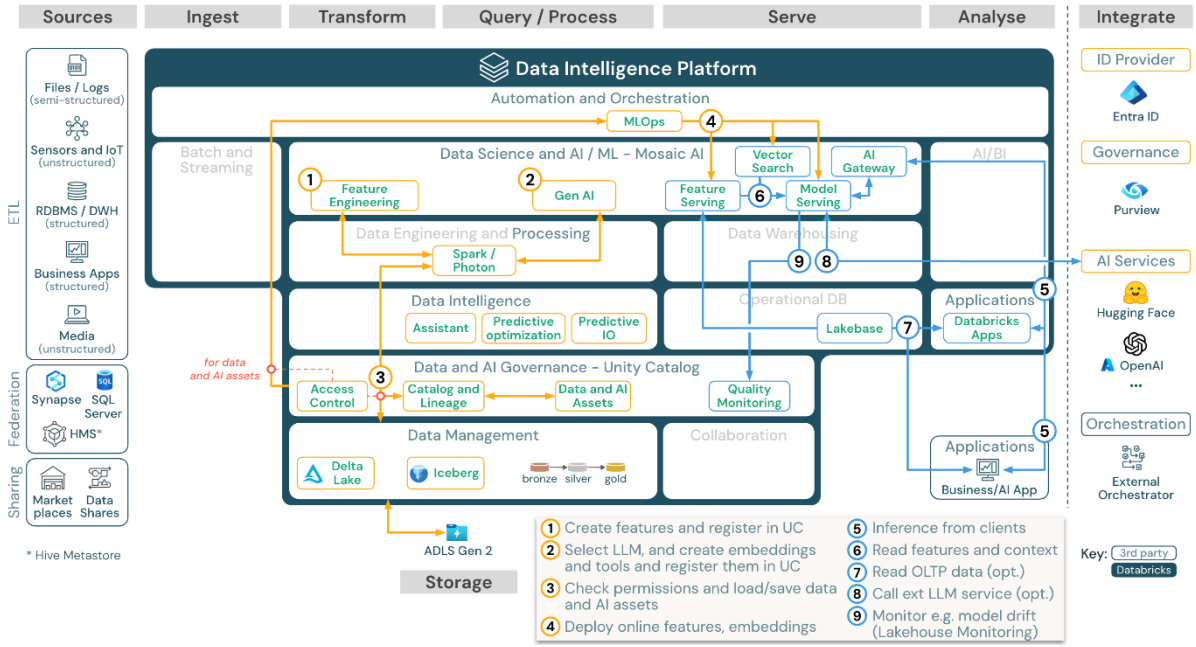
[Download: Machine learning and AI reference architecture for Azure Databricks]

For machine learning, the Databricks Data Intelligence Platform provides Mosaic AI, which comes with state-of-the-art machine and deep learning libraries. It provides capabilities such as Feature Store and Model Registry (both integrated into Unity Catalog), low-code features with AutoML, and MLflow integration into the data science lifecycle.

All data science-related assets (tables, features, and models) are governed by Unity Catalog and data scientists can use Lakeflow Jobs to orchestrate their jobs.

For deploying models in a scalable and enterprise-grade way, use the MLOps capabilities to publish the models in model serving.

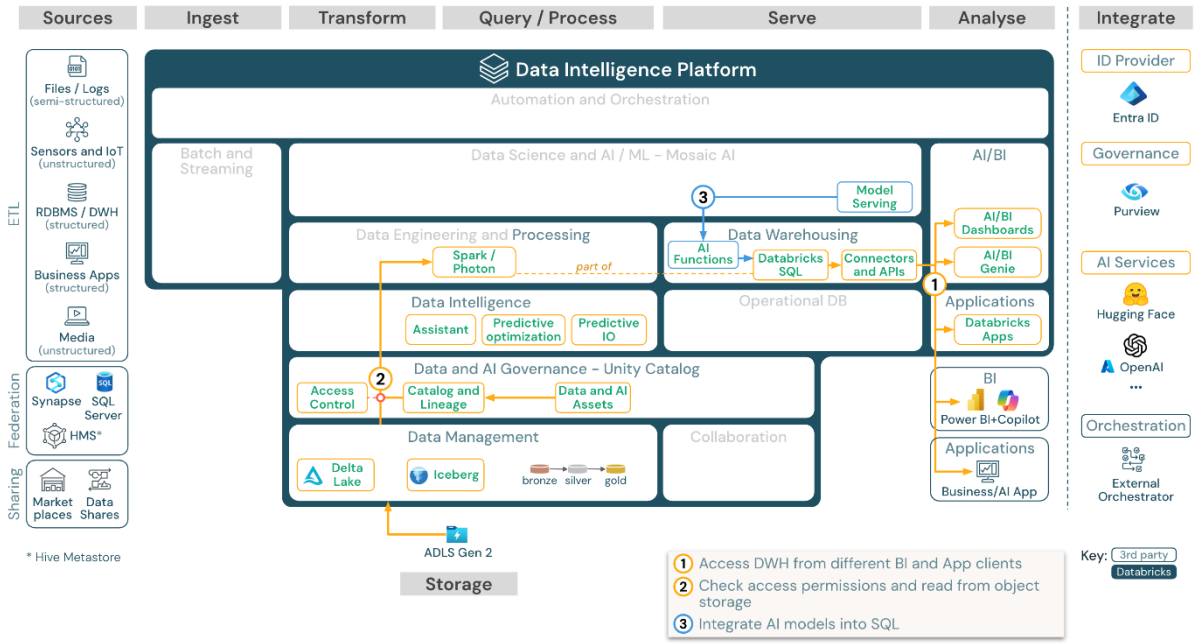
AI Agent applications (Gen AI)



[Download: Gen AI application reference architecture for Azure Databricks]

For deploying models in a scalable and enterprise-grade way, use the MLOps capabilities to publish the models in model serving.

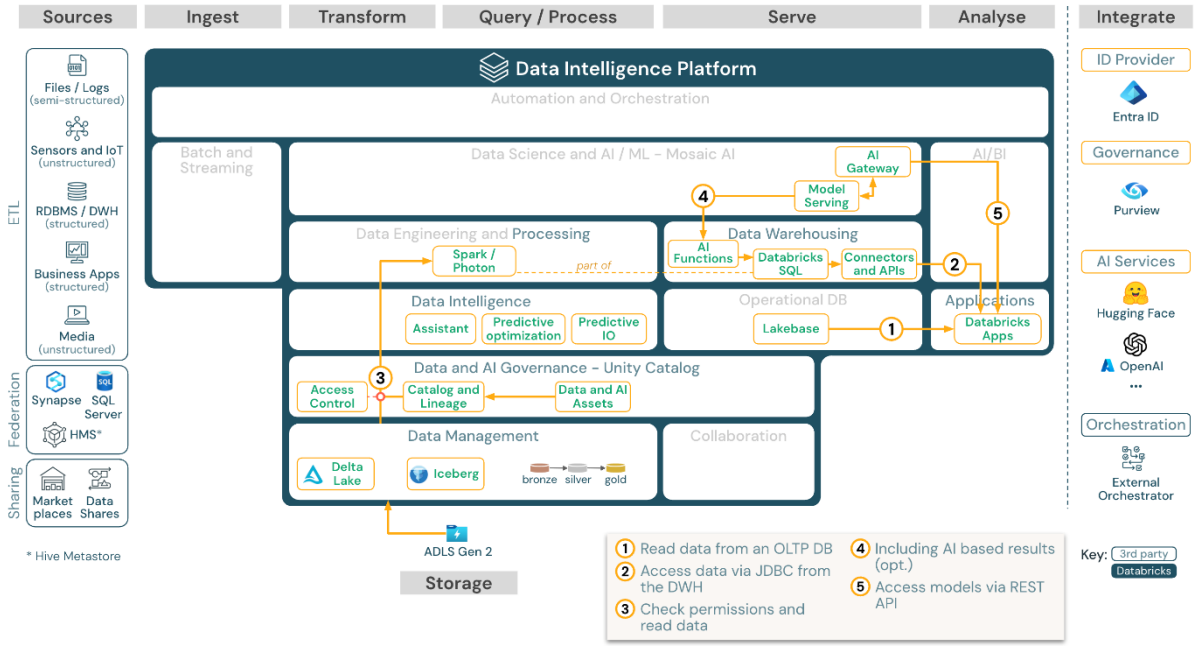
BI and SQL analytics



[Download: BI and SQL analytics reference architecture for Azure Databricks]

For BI use cases, business analysts can use dashboards, the Databricks SQL editor or BI tools such as Tableau or Power BI. In all cases, the engine is Databricks SQL (serverless or non-serverless), and Unity Catalog provides data discovery, exploration, and access control.

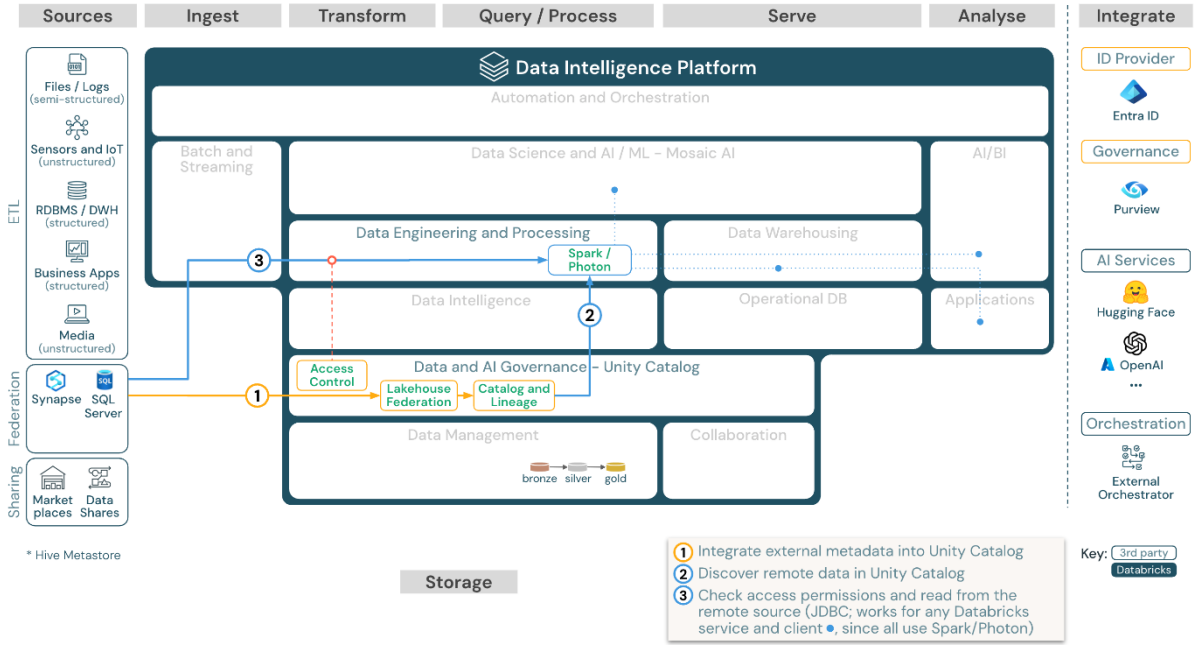
Business Apps



Download: Business Apps for for Databricks for Azure Databricks

Databricks Apps enables developers to build and deploy secure data and AI applications directly on the Databricks platform, which eliminates the need for separate infrastructure. Apps are hosted on the Databricks serverless platform and integrate with key platform services. Use Lakebase, if the app needs OLTP data that got synched from the Lakehouse.

Lakehouse federation

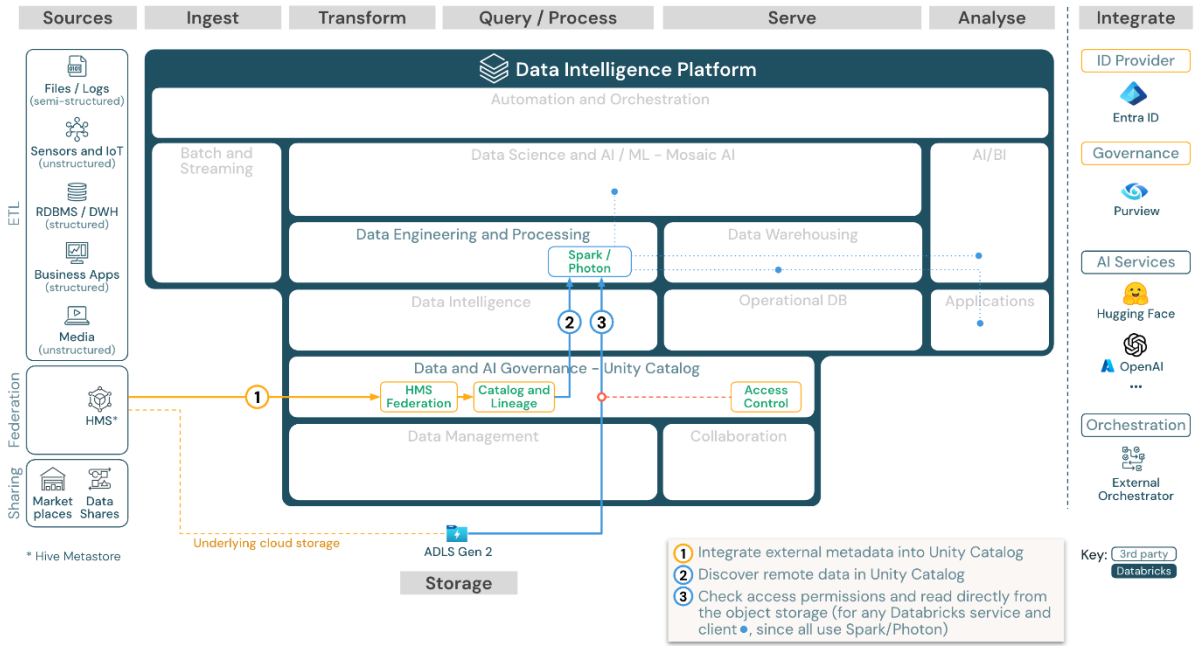


Download: Lakehouse federation reference architecture for Azure Databricks

Lakehouse Federation allows external data SQL databases (such as MySQL, Postgres, SQL Server, or Azure Synapse) to be integrated with Databricks.

All workloads (AI, DWH, and BI) can benefit from this without the need to ETL the data into object storage first. The external source catalog is mapped into the Unity catalog and fine-grained access control can be applied to access via the Databricks platform.

Catalog federation

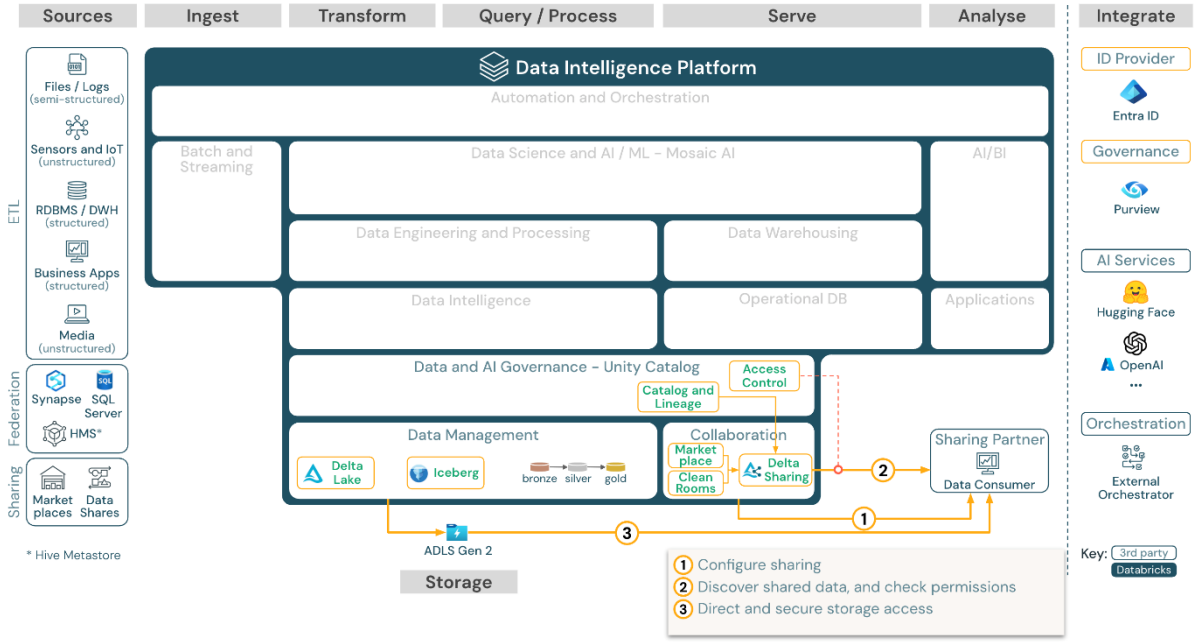


[Download: Catalog federation reference architecture for Azure Databricks]

Catalog federation allows external Hive Metastores (such as MySQL, Postgres, SQL Server, or Azure Synapse) to be integrated with Databricks.

All workloads (AI, DWH, and BI) can benefit from this without the need to ETL the data into object storage first. The external source catalog is added to Unity Catalog where fine-grained access control is applied via the Databricks platform.

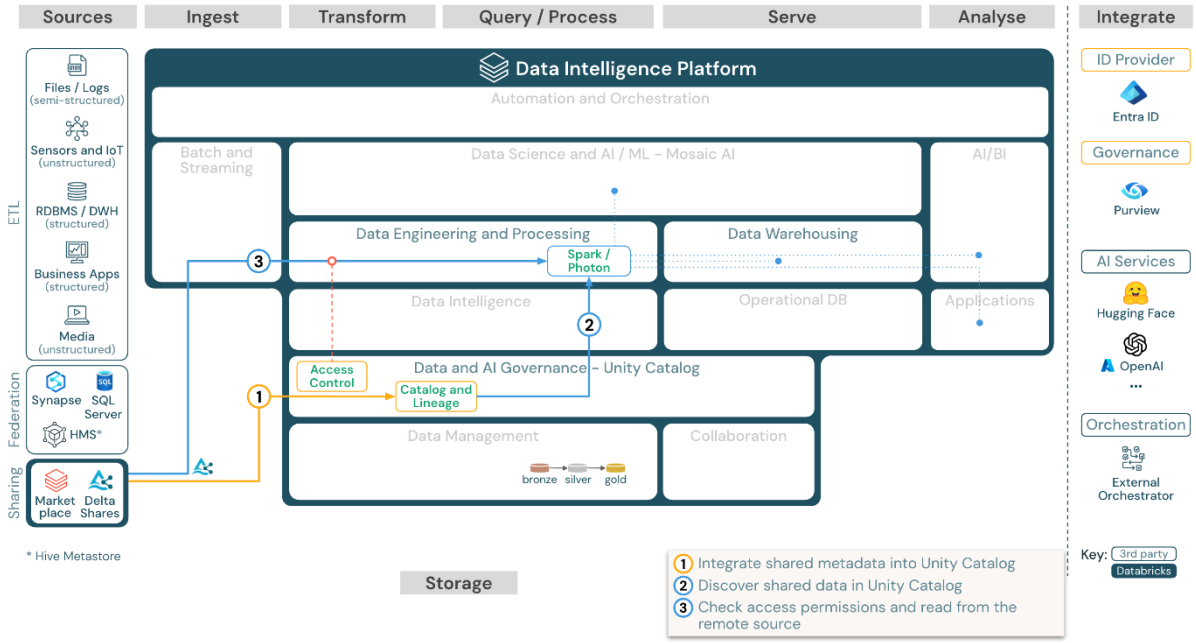
Share Data with 3rd party tools



[Download: Share data with 3rd-party tools reference architecture for Azure Databricks]

Enterprise-grade data sharing with 3rd parties is provided by Delta Sharing. It enables direct access to data in the object store secured by Unity Catalog. This capability is also used in the Databricks Marketplace, an open forum for exchanging data products.

Consume shared data from Databricks



[Download: Consume shared data from Databricks reference architecture for Azure Databricks]

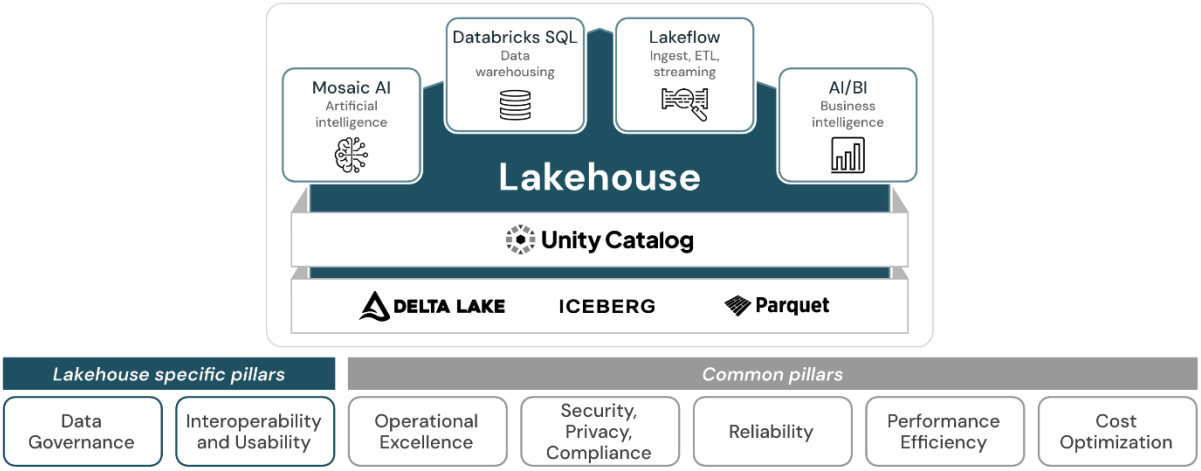
The Delta Sharing Databricks-to-Databricks protocol allows to share data securely with any Databricks user, regardless of account or cloud host, as long as that user has access to a workspace enabled for Unity Catalog.

Data lakehouse architecture: Databricks well-architected framework

- 04/18/2025

This set of data lakehouse architecture articles provides principles and best practices for the implementation and operation of a lakehouse using Azure Databricks.

Databricks well-architected framework for the lakehouse



The *well-architected lakehouse* consists of 7 pillars that describe different areas of concern for the implementation of a data lakehouse in the cloud:

- **Data and AI governance**

The oversight to ensure that data and AI bring value and support your business strategy.

- **Interoperability and usability**

The ability of the lakehouse to interact with users and other systems.

- **Operational excellence**

All operations processes that keep the lakehouse running in production.

- **Security, privacy, and compliance**

Protect the Azure Databricks application, customer workloads, and customer data from threats.

- **Reliability**

The ability of a system to recover from failures and continue to function.

- **Performance efficiency**

The ability of a system to adapt to changes in load.

- **Cost optimization**

Managing costs to maximize the value delivered.

The *well-architected lakehouse* extends the Microsoft Azure Well-Architected Framework to the Databricks Data Intelligence Platform and shares the pillars "*Operational Excellence*," "*Security*" (as "*Security, privacy, and compliance*"), "*Reliability*," "*Performance Efficiency*," and "*Cost Optimization*."

For these five pillars, the principles and best practices of the cloud framework still apply to the lakehouse. The *well-architected lakehouse* extends these with principles and best practices specific to the lakehouse and important to build an effective and efficient lakehouse.

The lakehouse-specific pillars

The pillars "*Data and AI Governance*" and "*Interoperability and Usability*" cover concerns specific to the lakehouse.

Data and AI governance encapsulates the policies and practices implemented to securely manage the data and AI assets within an organization. One of the fundamental aspects of a lakehouse is centralized data and AI governance: The lakehouse unifies data warehousing and AI use cases on a single platform. This simplifies the modern data stack by eliminating the data silos that traditionally separate and complicate data engineering, analytics, BI, data science, and machine learning. To simplify these governance tasks, the lakehouse offers a unified governance solution for data, analytics and AI. By minimizing the copies of your data and moving to a single data processing layer where all your data and AI governance controls can run together, you improve your chances of staying in compliance and detecting a data breach.

Another important tenet of the lakehouse is to provide a great user experience for all the personas that work with it, and to be able to interact with a wide ecosystem of external systems. Azure already has a variety of data tools that perform most tasks a data-driven enterprise might need. However, these tools must be properly assembled to provide all the functionality, with each service offering a different user experience. This approach can lead to high implementation costs and typically does not provide the same user experience as a native lakehouse platform: Users are limited by inconsistencies between tools and a lack of collaboration capabilities, and often have to go through complex processes to gain access to the system and thus to the data.

An integrated lakehouse on the other side provides a consistent user experience across all workloads and therefore increases usability. This reduces training and onboarding costs and improves collaboration between functions. In addition, new features are automatically added over time - to further improve the user experience - without the need to invest internal resources and budgets.

A multi-cloud approach can be a deliberate strategy of a company or the result of mergers and acquisitions or independent business units selecting different cloud providers. In this case, using a multi-cloud lakehouse results in a unified user experience across all clouds. This reduces the proliferation of systems across the enterprise, which in turn reduces the skill and training requirements of employees involved in data-driven tasks.

Finally, in a networked world with cross-company business processes, systems must work together as seamlessly as possible. The degree of interoperability is a crucial criterion here, and the most recent data, as a core asset of any business, must flow securely between internal and external partners' systems.